Universität der Bundeswehr München

Masterarbeit

zur Erlangung des akademischen Grades

Master of Engineering

im Studiengang

Computer Aided Engineering

Entwicklung von Features zur Verzahnung von labAlive Web- und App und Suchfunktion

Leutnant Robin Herbst (B. Eng.)

Computer Aided Engineering

Jahrgang 2022

Matrikelnummer: 1197319

Betreuender Hochschullehrer

Prof. Dr. Erwin Riederer

ETTI 5 - Institut für Funkkommunikation

22. August 2023

Erklärung

“Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen.”

....

Ort, Datum Unterschrift

“Der Speicherung meiner Studienarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.“

....

Ort, Datum Unterschrift

Robin Herbst

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

IDE Integrated Development Environment

Kap. Kapitel

Abb. Abbildung

JSP Jakarta Server Pages

URI Uniform Resource Identifier

HTML Hypertext Markup Language

CSS Cascading Style Sheets

z.B. zum Beispiel

bzw. beziehungsweise

Z. Zeile

Kapitel 1

Einleitung

In der Ära des digitalen Zeitalters spielen Webseiten eine entscheidende Rolle als primäre Informationsquelle und Plattformen für die Lehre.
Die Fülle an verfügbaren Informationen auf Websites stellt jedoch sowohl für Benutzer als auch für Website-Betreiber eine Herausforderung dar. Um den Besuchern eine nahtlose und zielgerichtete Erfahrung zu bieten und gleichzeitig das Potenzial der Webseite optimal auszuschöpfen, sind leistungsstarke Suchalgorithmen und Datenbanken unerlässlich.
Die vorliegende Masterarbeit widmet sich genau diesem Thema und untersucht, wie die Verwendung eines effizienten Suchalgorithmus in Kombination mit einer gut strukturierten Datenbank die Benutzererfahrung verbessert und die Leistung der Webseite optimiert.
Eine gut organisierte Datenbank ermöglicht es dem Suchalgorithmus, Suchanfragen schnell und effizient zu verarbeiten, ohne die Serverressourcen unnötig zu belasten. Ein intelligentes Datenbankdesign kann außerdem sicherstellen, dass die Website auch bei wachsendem Datenbestand eine optimale Leistung bietet.
Ziel dieser Masterarbeit ist es, den Suchalgorithmus zu optimieren und bereits bestehende Daten in ein neues Datenbanksystem zu integrieren um die Nutzererfahrung und Leistung einer Webseite eingehend zu verbessern.
Auch für Website-Betreiber soll diese Arbeit wertvolle Erkenntnis bieten, wie sie ihre Suchfunktionen und Datenbanken optimieren können, um die Zufriedenheit der Benutzer zu steigern und den Erfolg ihrer Webpräsenz nachhaltig zu fördern.

1.1 Motivation

Der wachsende Bedarf an einer präzisen und benutzerfreundlichen Suche auf Webseiten resultiert aus dem zunehmenden Umfang an Informationen, die im Internet verfügbar sind.
Ein schlecht optimierter Suchalgorithmus kann dazu führen, dass Benutzer sich in einem Meer von irrelevanten Ergebnissen verlieren, was wiederum zu einem vorzeitigen Verlassen der Webseite führen kann.
Auf der anderen Seite kann ein ausgefeilter Suchalgorithmus ein personalisiertes und reibungsloses Sucherlebnis bieten. Dies erhöht die Wahrscheinlichkeit, dass Besucher länger auf der Seite verweilen, sich tiefer mit den Inhalten auseinandersetzen und letztendlich zu aktiven Nutzern werden.
Nicht nur der Suchalgorithmus selbst, sondern auch die Anzeige und Darstellung der Suchergebnisse spielt eine äußerst wichtige Rolle in diesem Zusammenhang.
Ein guter und präziser Suchalgorithmus auf einer Webseite bietet zahlreiche Vorteile, die sowohl für die Benutzer als auch für die Website-Betreiber von großer Bedeutung sind. Hier sind einige der wichtigsten Vorteile:
- Schnellere und effizientere Suche: Ein guter Suchalgorithmus liefert schnell und effizient relevante Ergebnisse. Benutzer können ihre gewünschten Informationen schneller finden, was zu einer besseren Benutzererfahrung führt.
- Steigerung der Seitenaufrufe: Eine präzise Suchfunktion verleitet die Benutzer dazu, mehr Zeit auf der Website zu verbringen, da sie einfacher auf relevante Inhalte zugreifen können.
- Verbesserte Benutzererfahrung: Ein präziser Suchalgorithmus liefert genau die Ergebnisse, nach denen die Benutzer suchen, und reduziert die Anzahl irrelevanter oder unpassender Ergebnisse. Dies führt zu einer insgesamt besseren Benutzererfahrung und erhöht die Wahrscheinlichkeit, dass die Benutzer zufrieden sind und auf der Webseite bleiben.

1.2 Kurzfassung

Diese Masterarbeit besteht im wesentlichen aus vier Teilen:
1. Erweiterung und Verbesserung der Suchmaschine
2. Umstellung auf neues Datenmodell und Datenbank
3. Migration bestehender Daten ins neue Datenmodell
4. Übersichtliche und benutzerfreundliche Anzeige der Suchresultate.

Die Struktur dieser Arbeit zieht sich nahezu identisch durch das gesamte Dokument. Wenn Klassen erläutert werden, beginnt das Kapitel stets mit einem Klassendiagramm als Übersicht.
Das gesamte Projekt wird ebenfalls mittels verschiedener Klassendiagramme visualisiert. Daraufhin folgt die Beschreibung der Methoden. Hierbei werden ausschließlich die wichtigen oder komplexen Methoden im Detail erläutert. Im eigentlichen Text werden die Methoden in der Regel ohne ihre Parameter aufgeführt, da diese aus dem Abbild der Methode ersichtlich sind. Um den Überblick zu wahren, sind Methoden und Klassen fett und kursiv hervorgehoben, während Variablen lediglich kursiv formatiert sind.

Diese Masterarbeit baut auf einigen verwandten Arbeiten auf:
- Albert Lattke, Bachelorarbeit 2022 - Entwicklung Datenmodell (veraltet)
- Stephen Günter, Bachelorarbeit 2022 - Erstellung Images
- Johannes Krüger, Bachelorarbeit 2022 - Erstellung Signale
- Herbst Robin, Bachelorarbeit 2022 - Entwicklung Suchmaschine (veraltet)
- Niklas Dombek, Bachelorarbeit 2023 - Entwicklung neues Datenmodell
- Ludwig Stöckl, Bachelorarbeit 2023 - Entwicklung neues Datenmodell

Um zukünftigen Entwicklern eine effiziente und umfassende Einsicht in das Projekt zu ermöglichen, wurden für einen Großteil des neuen Datenmodells und der entwickelten Suchmaschine mehrere Klassendiagramme erstellt. Diese Diagramme sollen für eine schnelle Orientierung in zukünftigen Projekten helfen.
[Abb.A.4] - package resource.data (ResourceData minimiert)
[Abb.A.5] - Class ResourceData
[Abb.A.6] - package resource.additional
[Abb.4.1] - package searchEngine

Kapitel 2

Verwendete Software

2.1 Eclipse

Eclipse IDE wird häufig in verschiedenen Projekten eingesetzt, darunter auch Webentwicklung. Aufgrund seiner Flexibilität, Erweiterbarkeit und großen Entwicklergemeinschaft bleibt Eclipse eine der führenden IDEs für Softwareentwicklungsaufgaben und bietet eine robuste Umgebung für Entwickler, um qualitativ hochwertige Anwendungen zu erstellen.

2.2 IntelliJ

IntelliJ IDEA hat sich aufgrund seiner hohen Benutzerfreundlichkeit als eine der führenden IDEs etabliert. Es wird von vielen Entwicklern auf der ganzen Welt als bevorzugte Entwicklungsumgebung für Java- und andere Softwareprojekte eingesetzt.

2.3 Git

GIT ermöglicht es Entwicklern, den Verlauf von Änderungen an ihrem Code effizient zu verfolgen. GIT erleichtert die Zusammenarbeit in Teamprojekten, da mehrere Entwickler gleichzeitig an verschiedenen Teilen des Projekts arbeiten können. Sie können ihre Änderungen in separaten Branches verwalten. Außerdem ermöglicht es im Falle von Entwicklerfehlern, diese rückgängig zu machen und auf den vorherigen Stand zurückzuspringen.

Die Entscheidung, für dieses Projekt GIT zu verwenden, brachte sowohl Vor- als auch Nachteile mit sich. Die Möglichkeit, eigenständig von zu Hause aus zu arbeiten und stets Zugriff auf den aktuellen Code zu haben, erwies sich als äußerst praktisch. Allerdings erforderte es auch eine gewisse Zeitspanne, um sich mit den verschiedenen Funktionen vertraut zu machen.

Kapitel 3

Allgemeines über labAlive

Die Webseite **labAlive** [2] bietet eine faszinierende Möglichkeit, sich interaktiv mit verschiedenen Aspekten der Kommunikationstechnologie auseinanderzusetzen.
Die Webseite dient als virtuelles Labor, in dem Nutzer Experimente im Bereich der Kommunikationstechnik durchführen können, ohne physische Ausrüstung oder Laborzugang zu benötigen.
labAlive stellt eine breite Palette an Experimenten zur Verfügung, die es den Nutzern ermöglichen, die Grundlagen und fortgeschrittenen Konzepte der Kommunikationstechnologie zu erforschen.
Von der Übertragung von Daten bis hin zur Untersuchung von Rauschen und Signalverarbeitung bietet die Plattform eine vielfältige Auswahl an Versuchen. Die Benutzeroberfläche von **labAlive** ist intuitiv gestaltet und ermöglicht es den Nutzern, Experimente einfach auszuwählen, durchzuführen und die Ergebnisse in Echtzeit zu beobachten. Jedes Experiment wird durch klare Anleitungen und Erklärungen begleitet, die sowohl für Einsteiger als auch für Fachleute verständlich sind.
Die Website bietet auch Möglichkeiten zur Anpassung von Parametern, so dass Nutzer verschiedene Szenarien simulieren können.
Durch neue Funktionen erhalten Nutzer die Möglichkeit, sich anzumelden und Experimente unter ihrem Profil zu erstellen, die sie anschließend speichern können. Diese Experimente können darüber hinaus veröffentlicht werden und stehen somit allen Nutzern zur Verfügung.

Kapitel 4

Übersicht SearchEngine

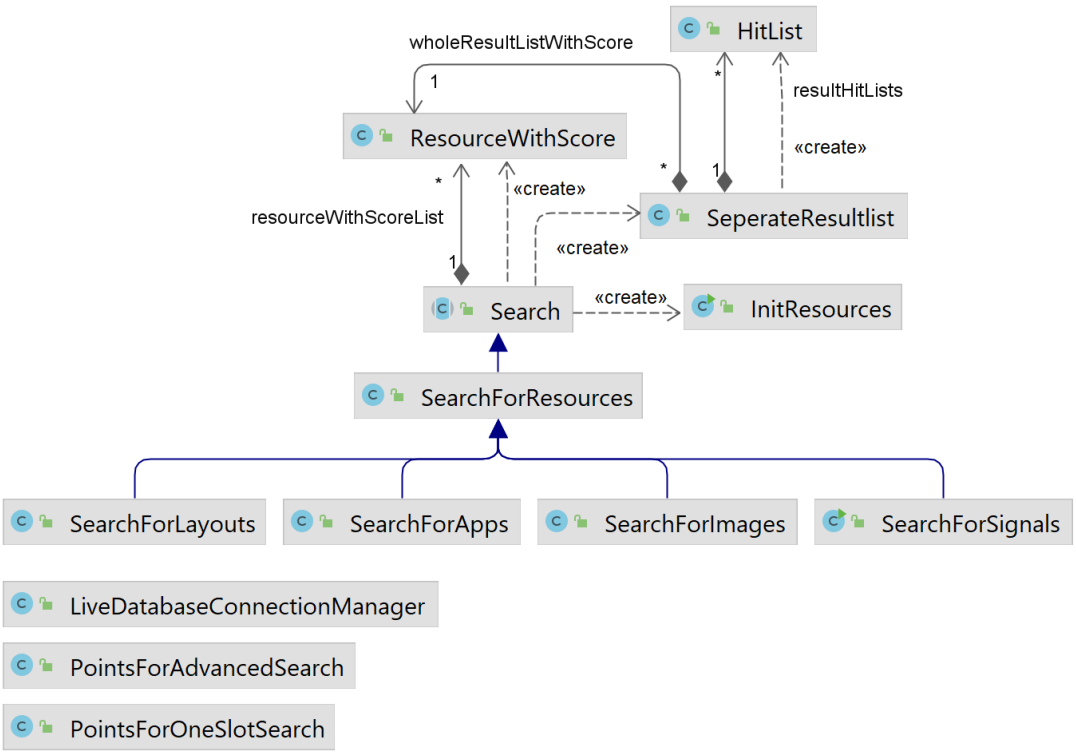

Abbildung 4.1: package searchEngine

Kapitel 5

Update und Erweiterung der SearchEngine

Um die nachfolgende Umstellung und Erweiterung der Suchmaschine zu erleichtern, wurde ihre Grundstruktur überarbeitet. Dadurch wurde sie sowohl für kommende Entwicklungen als auch für mögliche zukünftige Erweiterungen optimiert.
Durch die Integration der Ressourcen, Layouts und Wirings, wurde die Suchmaschine um zwei Ressourcen erweitert.

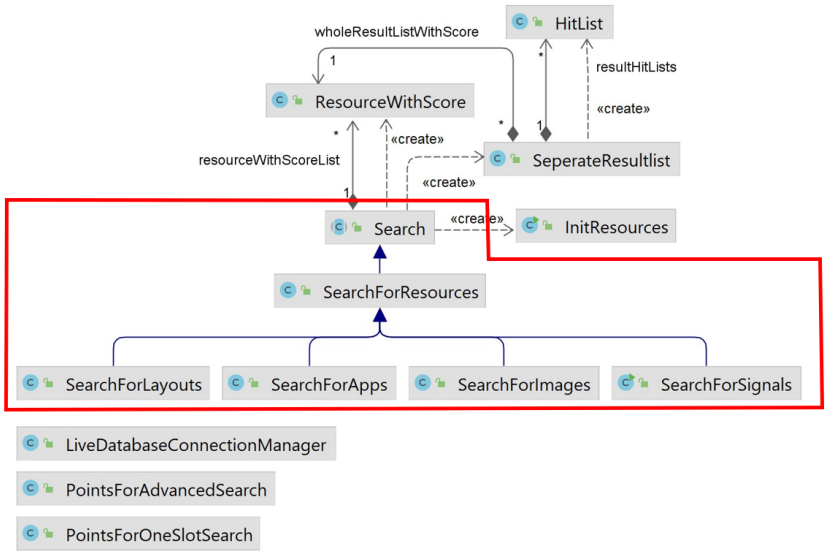

Abbildung 5.1: Übersicht Projekt, Suche

Der Plan sah vor, die bestehende Suchmaschine in mehrere Klassen aufzuteilen und die einzelnen Ressourcensuchen in separate Klassen zu gliedern. Wie aus dem Klassendiagramm [Abb.5.1] ersichtlich ist, erfolgt die eigentliche Suche lediglich in den Klassen ***Search***, ***SearchForResources***, ***SearchForLayouts***, ***SearchForApps***, ***SearchForImages*** und ***SearchForSignals***.
Der Ablauf jeder Suche startet in der Oberklasse Search, wo die Daten vorbereitet werden. Anschließend wird in der Klasse SearchForResources nach den Ressourcenparametern gesucht, gefolgt von der Suche in den spezifischen Klassen nach deren Parametern.
Die Klassen für die Punktzahlen (PointsForAdvancedSearch und PointsForOneSlotSearch) blieben unverändert.

5.1 Update der SearchEngine

5.1.1 Class Search

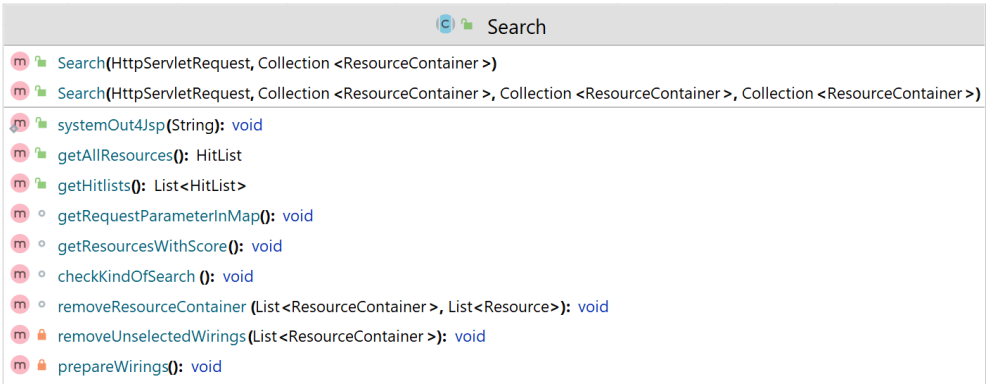

Abbildung 5.2: Class Search

Diese Oberklasse bereitet mit ihren verschiedenen Funktionen und Listen die Suche der eigentlichen „Suchklassen“ vor. Die Parameter zur Einstellung der Suchschärfe befinden sich ebenfalls in dieser Klasse und sind nahezu identisch mit der veralteten Suchmaschine. Konstruktoren unterscheiden sich nur in ihrer Länge, da die Suche nach verschiedenen Wirings in derselben JSP erfolgt, aber aus drei unterschiedlichen Wiringobjekten zusammengesetzt ist.
Die Methode ***removeResourceContainer()*** wird verwendet um die Kapselung eines Containers um eine Ressource zu entfernen und dann in die entsprechende Suchliste einfügt. Die Art der Suche (Einschlitz oder Erweitert) wird in der Methode ***checkKindOfSearch()*** bestimmt und beruht sich auf das Ergebnis von ***getRequestParameterInMap()***, welche im Folgenden genauer beschrieben wird. Nach Beendigung der Suche kann dann mit ***getHitlists()*** auf die Lösungslisten [Kap.8] zugegriffen werden.
In den jeweiligen JSPs wird sowohl der HttpServletRequest als auch die zu suchenden Ressourcen in den Konstruktor übergeben. Aus diesem HttpServletRequest werden die Parameternamen und deren Wert mit Hilfe der Methode ***getRequestParameterInMap()*** in eine HashMap (*requestParameter*) geschrieben.

Als Beispiel wird ein kurzer Ausschnitt einer URI präsentiert [Abb.5.3]. Es sind sowohl gefüllte (grün) als auch ungefüllte (rot) Parameter enthalten.

Abbildung 5.3: URI

```
87  void getRequestParameterInMap() {
88      Enumeration<String> paramNames = request.getParameterNames();
89      while (paramNames.hasMoreElements()) {
90          String paramName = paramNames.nextElement();
91          //gets only filled parameters!
92          if (Urls.parameterIsSet(request, paramName)) {
93              requestParameter.put(paramName, request.getParameter(paramName));
94          }
95      }
96      // s-->resource
97      if (Urls.parameterIsSet(request, "s")) {
98          kindOfResource = request.getParameter("s");
99          requestParameter.remove("s");
100     }
101     // q-->oneSlot
102     if (Urls.parameterIsSet(request, "q")) {
103         searchString = request.getParameter("q");
104         isOneslotInput = true;
105     }
106 }
```

Abbildung 5.4: getRequestparameterInMap()

Mit der Funktion ***getParameterNames()*** in Zeile 88 werden alle Parameternamen in einer Enumeration aus der URI zurückgegeben. Anschließend wird geprüft ob dieser einen gültigen Wert besitzt. Ist dies der Fall so wird Name und der gültige Wert des Parameters in einer HashMap verlinkt.
Danach werden die Parameter „s“ und „q“ einzeln behandelt. Der Parameter „s“ gibt an, nach welcher Art von Ressourcen gesucht werden soll. Sobald dieser in der Variablen *kindOfResource* gespeichert wurde, muss er aus der Parametermap entfernt werden.
Diese Aktion ist wichtig um später prüfen zu können ob ein Benutzer überhaupt Angaben in der Suchmaske getätigt hat. Wenn dieser nicht entfernt wird denkt die Logik, dass ein Such-Parameter gesetzt ist und setzt unnötigerweise die Suche an.

Der “q“-Parameter ist wie so häufig für die Einschlitzsuche verantwortlich und speichert ihren Wert global in *searchString*. Wenn dieser gesetzt ist, wird der boolesche Wert der Variablen *isOneslotInput* auf true gesetzt. Dies bedeutet im Umkehrschluss, dass die Einschlitzsuche der erweiterten Suche vorgezogen wird, wenn alle Parameter gesetzt würden.

```
{description=noise, id=156, title=Gauss}
```

Abbildung 5.5: filledRequestMap

Das Ergebnis der übergebenen URI [Abb.5.3] ist in der globalen Hashmap *requestParameter* dargestellt. Diese wird später von den spezifischen Unterklassen für die Suche verwendet. Wie zu sehen ist sind dort jetzt nur die gefüllten Parameter gespeichert. Auch der “s“-Parameter ist hier nicht zu finden.

```
79 void checkKindOfSearch() {
80     if (isOneslotInput) {
81         kindOfSearch = "oneslot";
82     } else if (!requestParameter.isEmpty())
83         kindOfSearch = "advanced";
84 }
```

Abbildung 5.6: checkKindOfSearch()

Aus den gewonnen Ergebnissen wird dann die Art der Suche festgestellt. Wenn der Einschlitzparameter gesetzt ist wird die Einschlitzsuche begonnen.

Wenn dagegen der Einschlitzparameter nicht gesetzt ist und die request-Hashmap nicht leer ist wird die erweiterte Suche angekurbelt.
Wenn keiner dieser Fälle vorliegt so haben Benutzende keinen Parameter in der Suchmaske eingegeben. In diesem Fall werden alle Ressourcen angezeigt. Dieser Vorgang ist zudem für die Verteilung der Punktzahl verantwortlich, da eine unterschiedliche Anzahl an Punkten für die Suchen verteilt werden.

Die nachfolgende Methode ***prepareWirings()*** wird nur im Konstruktor für die Suche nach Wirings (Apps) aufgerufen. Da diese Suche aus drei unterschiedlichen Wiringobjekten besteht werden diese zusammengeführt.

Um die Benutzererfahrung und Suche zu verbessern, kann auch nach den einzelnen Wiringobjekten gefiltert werden. In der Suchmaske für Apps kann dann durch ein einfaches Filtersystem [Abb.5.7] auf die einzelnen Objekte zugegriffen werden.

Abbildung 5.7: select Wirings

```
107  private void prepareWirings() {
108      removeResourceContainer(allDbRunWiringsContainer, resourceList);
109      removeResourceContainer(allDbWiringsContainer, resourceList);
110      removeResourceContainer(allDbQueryStringsContainer, resourceList);
111
112      boolean searchForMyWiringApps = false;
113      boolean searchForQueryStringWiringApps = false;
114      boolean searchForRunWiringApps = false;
115      //MyWiring
116      if (Urls.parameterIsSet(request, "myWiringApps")) {
117          requestParameterNames.remove("myWiringApps");
118          searchForMyWiringApps = true;
119      }
120      //QueryString
121      if (Urls.parameterIsSet(request, "queryStringWiringApps")) {
122          requestParameterNames.remove("queryStringWiringApps");
123          searchForQueryStringWiringApps = true;
124      }
125      //RunWiring
126      if (Urls.parameterIsSet(request, "runWiringApps")) {
127          requestParameterNames.remove("runWiringApps");
128          searchForRunWiringApps = true;
129      }
130      //remove non-selected Resources
131      //if non is selected search for all
132      if (searchForMyWiringApps || searchForQueryStringWiringApps
133              || searchForRunWiringApps) {
134
135          if (!searchForMyWiringApps) {
136              removeUnselectedWirings(allDbWiringsContainer);
137          }
138          if (!searchForQueryStringWiringApps) {
139              removeUnselectedWirings(allDbQueryStringsContainer);
140          }
141          if (!searchForRunWiringApps) {
142              removeUnselectedWirings(allDbRunWiringsContainer);
143          }
```

Abbildung 5.8: prepareWirings()

Zu Beginn werden die im Konstruktor übergebenen Collections von ihren Containern getrennt und in die entsprechende *resourceList* übertragen. Danach wird jeder Parameter für die zu Suchenden Objekte geprüft. Ist dieser vorhanden so werden deren boolesche Werte auf true gesetzt und anschließend aus der *requestParameter*-Map gelöscht. Dieser Vorgang hat wie in [Abb.5.4] erklärt eine wichtige Bedeutung für den weiteren Suchverlauf.

Zuletzt wird mit einer IF-Abfrage geprüft, ob überhaupt eines der in [Abb.5.7] zu selektierenden Objekte ausgewählt wurde. Ist dies nicht der Fall, so wird trotzdem nach allen Objekten gesucht.
Wenn doch selektiert wurde, werden nicht-selektierte Ressourcen wieder aus der *resourceList* entfernt. Die Bedeutung der negativ-Logik ist in diesem Fall wichtig, da bei der Einschlitzsuche immer alle Ressourcen durchsucht werden müssen. Des Weiteren ist es für Benutzende leichter verständlich wenn zu Beginn in der Suchmaske direkt alle Ressourcen bereits ausgewählt sind.

5.1.2 Class SearchForResources

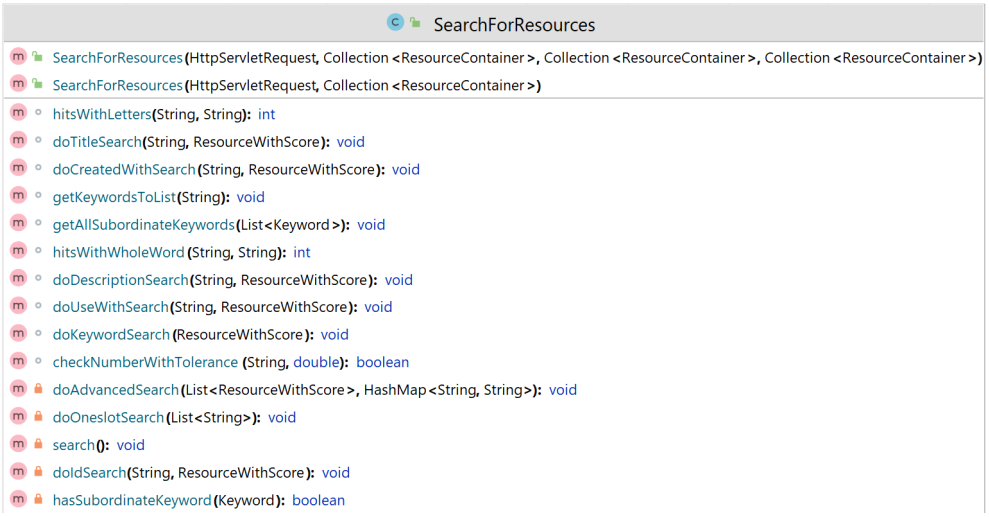

Abbildung 5.9: Class SearchForResources

Die Klasse SearchForResources ist die erste Klasse, in welcher die aktive Suche stattfindet. Sie enthält den Hauptbestandteil aus der veralteten Suchmaschine. Unter anderem die wichtigsten Suchfunktionen wie ***hitsWithLetters(String, String)***, ***hitWithWholeWord(String, String)*** oder ***checkNumberWithTolerance(String, double)***. Diese Methoden bilden bei jedem Aufruf die eigentliche Suche. Auch die Suche nach Titel, Beschreibung und Schlüsselwörter ist dieselbe geblieben. Da die Struktur der Methoden nicht verändert wurde, gibt es keine genauere Beschreibung. Diese Methoden wurden in der Bachelorarbeit [1] erklärt.

Jede dieser Suchklassen ist nun einfach und verständlich aufgebaut. Somit kann sie leicht für erneute Erweiterungen in der Zukunft angepasst werden.

```
12  final private List<String> resourceParameter =
13          Arrays.asList(
14          "title",
15          "description",
16          "createdWith",
17          "usedWith",
18          "keywords",
19          "id");
```

Abbildung 5.10: Parameter für Ressourcen

Am Anfang der Klasse befinden sich in einem Array alle Parameter, nach welchen gesucht werden kann [Abb.5.10]. In der Klasse SearchForResources wird nur nach den Parametern gesucht, welche alle Ressourcen gemeinsam besitzen. Es ist wichtig, dass diese genauso heißen, wie in der Uri [Abb.5.3] übergeben.

```
21  public SearchForResources(HttpServletRequest request,
22          Collection<ResourceContainer> resourceDbCollection) {
23      super(request, resourceDbCollection);
24      getRequestParameterInMap();
25      checkKindOfSearch();
26      search();
27  }
```

Abbildung 5.11: Konstruktor SearchForResources

Im Konstruktor der Klasse wird durch den *super()*-Aufruf der Request und die zu durchsuchende Collection verarbeitet. Des Weiteren werden die Methoden ***getRequestParameterInMap()***,***checkKindOfSearch()*** und ***search()*** aufgerufen, welche in [Kap.5.2] erläutert wurden.
Bevor in ***search()*** die Suche gestartet wird, holt sich die Klasse alle aktiven Ressourcen und bindet diese an einen Score (*ResourceWithScore*).
Der Konstruktor für die Suche nach Wirings ist diesem, außer der Anzahl an Collections, gleich.
Die Erweiterung der Suchmaschine bestand auch darin, die Suche nach einer Ressourcen-ID, createdWith- und useWith-Ressourcen zu ermöglichen. Aufgrund der Vereinfachung der Suchmaschine konnten diese Parameter ohne Probleme eingefügt werden.
Folgend wird die erweiterte Suche beschrieben.

```
55  private void doAdvancedSearch(List<ResourceWithScore> resourceWithScoreList,
56                                HashMap<String, String> requestParameter) {
57      // Iterate through resources
58      for (ResourceWithScore activeResource : resourceWithScoreList) {
59          // Iterate through requestParameter
60          for (String paramName : requestParameter.keySet()) {
61              switch (paramName) {
62                  case "id": {
63                      doIdSearch(requestParameter.get(paramName), activeResource);
64                      break;
65                  }
66                  case "title": {
67                      doTitleSearch(requestParameter.get(paramName), activeResource);
68                      break;
69                  }
70                  case "description": {
71                      doDescriptionSearch(requestParameter.get(paramName), activeResource);
72                      break;
73                  }
74                  case "createdWith": {
75                      doCreatedWithSearch(requestParameter.get(paramName), activeResource);
76                      break;
77                  }
78                  case "usedWith": {
79                      doUseWithSearch(requestParameter.get(paramName), activeResource);
80                      break;
81                  }
82                  case "keywords": {
83                      getKeywordsToList(requestParameter.get(paramName));
84                      doKeywordSearch(activeResource);
85                      break;
86                  }
87                  case "uuid": {
88                      Session.instance(request).setUuid(requestParameter.get(paramName));
89                      break;
90                  }
91                  default: {
```

Abbildung 5.12: SearchForResources doAdvancedSearch()

Die Methode ***doAdvancedSearch()*** bekommt alle zu durchsuchenden Ressourcen mit deren Score. Zudem die *requestParameter*-Map [Abb.5.5]. Mit der ersten FOR-Schleife werden diese Ressourcen durchlaufen. In der darauffolgenden Schleife das Key-Set(Parameternamen) der *requestParameter*-Map. Somit wird sichergestellt, dass nur dort gesucht wird, wo die Suche auch erwünscht ist. Dies spart sowohl Ressourcen als auch Zeit. In den einzelnen Cases kann dann die spezifische Suche begonnen werden. Ein wichtiger Punkt in dieser Methode ist, dass eine Ressource hier niemals zur Ergebnisliste hinzugefügt werden darf. Die globale Liste *resultListWithScore* wird von den nachfolgenden Klassen ebenfalls verwendet. Wenn in dieser Klasse bei einem Treffer die Ressourcen direkt in die Ergebnisliste eingefügt würden, dann werden sie bei einem weiteren Treffer, in den nachfolgenden Klassen, wieder hinzugefügt. Somit könnte es passieren, dass eine Ressource mehrmals mit unterschiedlicher Punktzahl in der Trefferliste steht. Deshalb wird nur der Score in die Ressource eingetragen und später wird dieser eventuell verändert. Erst am Schluss der Suche kann die Ressource zu einem möglichen Ergebnis zählen.

```
45  private void doOneslotSearch(List<String> resourceParameter) {
46      requestParameter.clear();
47      for (String s : resourceParameter) {
48          requestParameter.put(s, searchString);
49      }
50      doAdvancedSearch(resourceWithScoreList, requestParameter);
51  }
```

Abbildung 5.13: SearchForResources doOneslotSearch()

Eigentlich stellte sich anfangs die Einschlitzsuche als komplexer und schwerer umzusetzen dar, da immer alle Ressourcenparameter durchsucht werden müssen. Durch die Eintragung der Ressourcenparameter [Abb.5.10] kann dies schnell und effektiv abgeleitet werden. Die Methode ***doOneslotSearch()*** wird mit diesen Ressourcenparameter aufgerufen.
In Zeile 46 wird die bestehende *requestParameter*-Map gelöscht. Anschließend werden alle Parameter mit dem Wert der Einschlitzsuche befüllt. Mit dieser neuen *requestParameter*-Map wird dann die ***doAdvancedSearch()*** durchgeführt.

```
q=bessel&s=signal&id=&title=&description=&createdWith=
```

```
12  final private List<String> resourceParameter =
13          Arrays.asList(
14          "title",
15          "description",
16          "createdWith",
17          "usedWith",
18          "keywords",
19          "id");
```

Abbildung 5.14: Einschlitz requestParameter

Kurz zusammengefasst ist die Einschlitzsuche nur eine erweiterte Suche mit allen zu suchenden Parametern befüllt mit dem *searchString* des “q“-Parameters. Die neue *requestParameter*-Map aus der Uri und den Parametern für eine Ressource [Abb.5.14] würde dann folgendermaßen aussehen:

```
description=bessel, usedWith=bessel, id=bessel, title=bessel}
```

Abbildung 5.15: Ausschnitt Parameter Einschlitzsuche

Wie am Anfang des Kapitels erklärt, werden nicht alle Methoden nochmals beschrieben. Für die Vollständigkeit wird jedoch noch eine spezifische Suche erklärt, welche neu hinzugefügt wurde.

```
98  private void doIdSearch(String searchWord, ResourceWithScore activeResource) {
99      try {
100         long resourceId = activeResource.getResource().getResourceID();
101         String[] searchWordList = searchWord.toLowerCase().split("\\s+");
102         for (String s : searchWordList) {
103             try {
104                 long searchID = Long.parseLong(s);
105                 if (searchID == resourceId) {
106                     activeResource.setIdScore(PointsForOneSlotSearch.ID);
107                 }
108             } catch (NumberFormatException e) {
109                 //System.out.println("no id-Format (long): "+s);
110             }
111         }
112     } catch (NullPointerException e) {
113         e.printStackTrace();
114     }
115 }
```

Abbildung 5.16: doIdSearch()

In der Theorie sind alle Methoden für die Suche nach bestimmten Ressourcenparameter identisch. Da sich jedoch häufig das Format ändert müssten zu viele verschiedene Fälle abgefangen werden. Auch die Fehlersuche ist einfacher. Die Erweiterung oder Änderung der Punktzahl fällt somit auch leichter und kann direkt an der Wurzel gepackt werden.
Gerade bei einer ID-Suche wird präzise gesucht und es dar keine Toleranz geben. Eine ID-Suche ist für Nutzende ein deutlicher Vorteil bei präzisen Suchanfragen.

5.2 Erweiterung der Ressourcen und deren Parameter

Im neuen Datenmodell gibt es momentan sieben unterschiedliche Ressourcen. Im Gegensatz zu der vorhandenen Suchmaschine ist dies eine Erweiterung um vier Ressourcen. Insgesamt kann nach Experimenten, Images, Layouts, Signalen, QueryStringWirings, RunWirings und Wirings gesucht werden. Bei den letzten drei Objekten handelt es sich allgemein um Wirings.
Die Suche nach diesen speziellen Objekten wird nachfolgend exemplarisch für Signale erklärt.

5.2.1 Class SearchForSignals (exemplarisch)

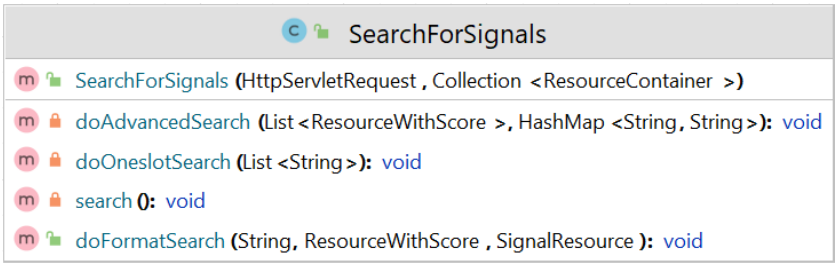

Abbildung 5.17: Class SearchForSignals

Eine der vier verschiedenen spezifischen Suchklassen wird hier anhand der Suche nach Signalen dargestellt. Der Konstruktor besteht nur aus einer Ressourcenkollektion. Auch in dieser Klasse gibt es die zwei unterschiedlichen Arten der Suche. Da die Einschlitzsuche ***doOneslotSearch()*** nur von den Ressourcenparametern abhängig ist, unterscheidet sich diese in keiner Klasse. Lediglich die erweiterte Suche ***doAdvancedSearch()*** ändert sich an einigen Stellen.

Die spezifischen Parameter für ein Signal werden genau wie in [Abb.5.10] beschrieben am Anfang der Klasse aufgelistet.

```
20  final private List<String> resourceParameter =
21          Arrays.asList(
22          "power",
23          "peakValue",
24          "paPr",
25          "numberOfSamples",
26          "sampleRate",
27          "format",
28          "length");
```

Abbildung 5.18: Parameter für Signale

Somit kann bei der erweiterten Suche auf jeden einzelnen Parameter zugegriffen werden.
Da es sich bei Signalparametern zum Großteil um eine Zahlensuche handelt, gibt es nur eine Methode für die Formatsuche.
Die Zahlensuche erfolgt direkt in der Switch-Case-Anweisung [Abb.5.19].

```
104      case "format": {
105           doFormatSearch(requestParameter.get(paramName),
106               activeResource, activeSignal);
107           break;
108           }
109      case "length": {
110           if (checkNumberWithTolerance(requestParameter.get(paramName),
111             activeSignal.getLenghInSec())) {
112             activeResource.addLengthScore(PointsForAdvancedSearch.NUMBERS);
113           }
114          break;
115         }
116      default: {
117          break;
118          System.out.println(paramName + ": not in switch-case:" + getClass());
119         }
120       }
121     }
122 } catch (ClassCastException | NullPointerException e) {
123     e.printStackTrace();
124 }
125 // if there are Hits then add to resultList
126 if (activeResource.getTotalScore() >= minScoreForResultList) {
127     resultListWithScore.add(activeResource);
128 }
```

Abbildung 5.19: SearchForSignals doAdvancedSearch()

Das Grundlegende Prinzip entspricht dem in [Abb.5.12]. Die Parameternamen werden durch die Switch-Case-Anweisung abgearbeitet. In Zeile 109 bis 114 ist ein Beispiel für der Zahlensuche dargestellt. Die Methode ***checkNumberWithTolerance()*** prüft direkt beide Werte und verteilt anschließend deren errechnete Punktzahl.
Der wesentliche Unterschied ist von Zeile 125 bis 128 erkennbar. Hier muss eine Ressource mit ihrem Score der Lösungsliste (resultListWithScore) hinzugefügt werden, wenn dieser über einer bestimmten Grenze (*minScoreForResultlist*) liegt. Somit kann der Administrator die Schärfe der Suche einstellen.

Für die anderen drei Suchklassen (*SearchForImages*, *SearchForLayouts* und *SearchForApps*) ist das Prinzip identisch. Es müssen nur deren Parameter angepasst werden. Anschließend muss ein Eintrag in der Switch-Case-Anweisung vorgenommen werden, welcher auf eine Suchmethode verweist. Somit ist in Zukunft gesichert, dass eine mögliche Erweiterung einfach umsetzbar wird.

Kapitel 6

Umstellung auf neues Datenmodell und Datenbank

Eine weitere Grundlegende Aufgabe dieser Arbeit beschäftigt sich mit der Umstellung des Datenmodells und die Anbindung an eine Datenbank. Die in den vorherigen Kapiteln beschriebene Erweiterung der Ressourcen hängt in direktem Zusammenhang mit dem neuen Datenmodell.

6.1 Vergleich Klassendiagramme

Für bessere Übersicht und Verständnis sind im Anhang Klassendiagramme aufgeführt. [Abb.A.4], [Abb.A.5] und [Abb.A.6] sind die wesentlichen Teile des neuen Datenmodells zusammengefasst.
[Abb.A.7] dagegen zeigt das alte Datenmodell. Es wird nicht genau auf die Änderungen im einzelnen eingegangen. Diese Klassendiagramme sollen lediglich dem Verständnis dienen.
Deutlich zu erkennen ist der unterschiedlich große Umfang der beiden Datenmodelle. Sowohl die Erweiterungen der Ressourcen als auch um einige Methoden. Durch die Vereinfachung der Suchmaschine [Kap.5] ist die Umstellung auf das neue Datenmodell im Bereich der Suchmaschine selbst nicht besonders schwer gefallen. Eines der größten Herausforderungen war es, dass einige Methoden nicht eins zu eins vom alten in das neue Datenmodell übernommen wurden. Somit mussten vor der Umstellung noch einige Änderungen vorgenommen werden, bevor bestehende Daten [Kap.7] in die Datenbank eingepflegt wurden.

6.2 Suchmasken

Durch das Hinzukommen neuer Ressourcen und Parameter muss auch die Oberfläche der Suchmaske spezifisch angepasst werden.

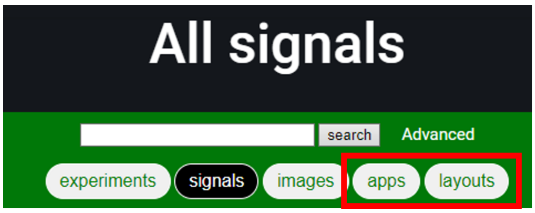

Abbildung 6.1: Übersicht Suchmaske

Wie in [Abb.6.1] zu sehen ist, sind zwei Reiter für die Suche nach Apps(Wirings) und Layouts hinzugekommen. Jeder dieser Buttons öffnet die spezifische JSP und startet die Suche.

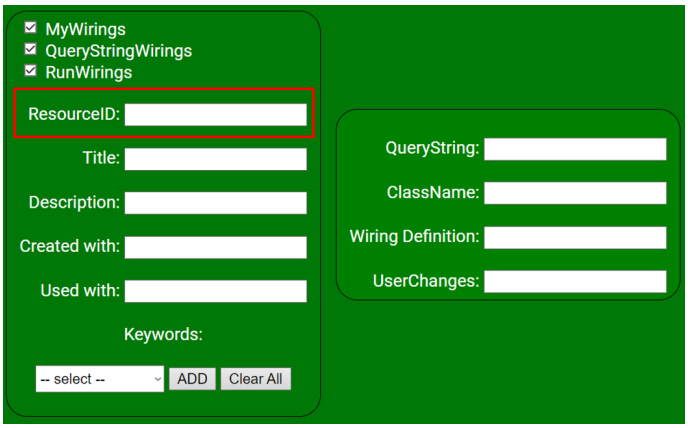

Abbildung 6.2: Suchmaske für Apps

Für alle Ressourcen wurde die Suche nach der Ressourcen-ID erweitert. Diese steigert die Benutzererfahrung deutlich. Durch die Suche nach einer eindeutigen ID kann der Suchvorgang schnell und präzise abgewickelt werden. Für Ressourcen-spezifische Anpassungen sind diese auf der rechten Seite zu finden. Auch die Selektion der verschiedenen Wirings [Abb.5.7] sind zu wählen. Haben Ressourcen wenige spezifische Ressourcenparameter, so werden diese nicht extra in einem neuen Raster aufgeführt, sondern unter dem *usedWith*-Feld eingefügt.

6.3 Detailseiten

Die Detailseiten sind für eine übersichtliche Anzeige der einzelnen Ressourcen verantwortlich. In ihnen werden alle wichtigen Parameter dargestellt. Jede Ressource hat ihre eigene JSP für die Anzeige. Der Aufbau dieser JSP´s ist nahezu identisch.

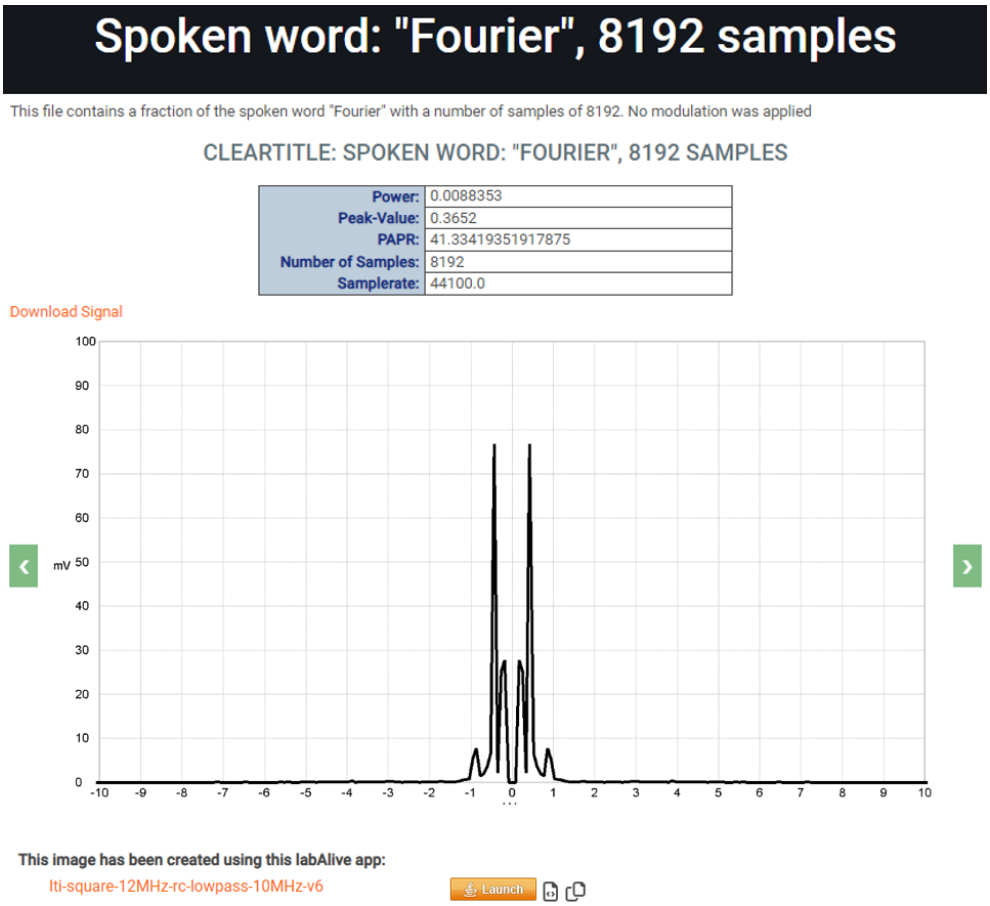

Abbildung 6.3: exemplarische Detailseite

Zu beginn erscheint in großer Schrift der Titel der jeweiligen Ressource. Direkt darunter folgt die Beschreibung. Dann sind bei Signalen deren spezifische Signalparameter in tabellarischem Aufbau zu sehen. Unmittelbar links oben neben dem Bild kann die Datei heruntergeladen werden. Eine Ressource kann sich in einer Kollektion befinden. Das bedeutet ähnliche Ressourcen werden zusammengeführt.

Ist eine Ressource in einer Kollektion so kann mit den Pfeilen links und rechts neben der Darstellung zwischen ähnlichen Ressourcen gewechselt werden. Um direkt zu einem möglichen Ursprung eines Bildes oder Signals zu kommen werden unter dem Bild noch verlinkte Ressourcen dargestellt. So können Verlinkungen direkt angeklickt oder sogar gestartet werden.

6.3.1 Aufbau Detail-JSP´s

Da in einer JSP Fehler nicht markiert werden, Java- und HTML-Code direkt ineinander überfließen und die Entwicklungsumgebung keine Vorschläge gibt, ist es besonders wichtig eine gute Struktur zu haben.

```
14 long id = Long.parseLong(request.getParameter("id"));
15 Resource resource = ResourceProvider.getResourceById(id).getResource();
16 String uuid = Session.instance(request).getUuid();
17 String titel = resource.getTitel();
18
19 if(!(resource.isPublic() || resource.isUsersResource(request))){
20     %>
21     <jsp:include page="/jsp/header.jsp" />
22     <section id="home_experiment">
23         <hr>
24         <h1>This Resource is not public or not yours. Logged in?</h1>
25     </section>
26     <%
27 }else{
```

Abbildung 6.4: Beginn signaldetails.jsp

Zuerst wird die Ressource-ID aus der URI gezogen und die entsprechende Ressource aus der Datenbank geladen. Dann wird direkt der Titel und die uuid in golbalen Variablen gespeichert. In Zeile 19 befinden sich zwei wichtige Prüfungen. Zuerst wird geprüft ob eine Ressource öffentlich ist. Gleichzeitig kann eine Ressource auch einem Benutzer gehören. Somit wäre die Einsicht auch erlaubt.

```
245 public boolean isUsersResource(HttpServletRequest request)
246     try{
247         Collection<ResourceContainer> usersResources =
248                 ResourceProvider.getAllResources(request);
249         for (ResourceContainer rc : usersResources){
250             if (rc.getResource().equals(this)){
251                 return true;
252             }
253         }
254     }catch (NullPointerException e){
255         System.out.println("Not logged in");
256     }
257     return false;
258 }
```

Abbildung 6.5: RescourceData isUsersResource()

Hierfür wird dann mit dem HTTPServletRequest der Benutzer ermittelt und all dessen Ressourcen mit der gesuchten verglichen [Abb.6.5]. Sollten also Benutzende versuchen eine private Ressource anzuschauen so wird ihnen auf der Detailseite angezeigt, dass diese nicht angemeldet sind oder die Ressource (noch) nicht vom Administrator publiziert wurde. Ist dies nicht der Fall so wird die Detailseite wie folgend erklärt aufgebaut.

```
30 <!-------------------Head----------------------------------->
31 %>
32 <jsp:include page="/jsp/header.jsp" />
33
34 <section id="home_experiment">
35     <hr>
36     <h1><%=  titel %></h1>
37
38 </section>
39 <section id="content">
40 <p><%=  resource.getDescription() %></p>
41
42 <%
43 if(!resource.getFormula().isEmpty()){
44     for(Formula formel : resource.getFormula()){
45         if(formel != null){%>
46
47             <h2><%="Formula: " + formel.getFormula()%></h2>
48
49     <%}}
50 }%>
51 <h2><%= "ClearTitle: " + titel%></h2>
52 <!-------------------Head END------------------------------->
```

Abbildung 6.6: Head signaldetails.jsp

Im Kopf der Detailseite wird dann der Titel und die Beschreibung angezeigt. Sollte die Ressource eine Formel besitzen so wird diese unmittelbar nach der Beschreibung dargestellt. Um an die spezifischen Signalparameter zu gelangen wird ein Class-cast vorgenommen und die Parameter in einer Tabelle dargestellt[Abb.6.3].

```
104 <!-------------------ContentImages-------------------------------->
105 <%
106 if(!resource.getContentImage().isEmpty() && !resource.isCollection()){
107 for(Image img : resource.getContentImage()){
108     %>
109     <img src=<%=img.getWholeContentImagePath()%> width="906" />
110     <%
111 }}
```

Abbildung 6.7: ContentImage signaldetails.jsp

Am Beispiel des ContentImages wird gezeigt wie zuerst geprüft wird ob der Parameter vorhanden ist und dann dementsprechend angezeigt[Abb.6.7]. Mit allen restlichen Parameter einer Ressource wird identisch verfahren [Abb.A.9].

Auf die Kollektion wird speziell eingegangen, da dort einiges beachtet werden muss.

Eine Kollektion kann z.B. aus den rechts aufgeführten Signalen bestehen, welche sich im Grunde nur in einem Parameter unterscheiden. Der Anfang wird Collection-Head genannt und wäre in diesem Fall ID vier.

ResourceID	Titel
4	Gaussian noise 1mVÂ²-2kHz
5	Gaussian noise 1mVÂ²-5kHz
6	Gaussian noise 1mVÂ²-10kHz
7	Gaussian noise 1mVÂ²-15kHz

Abbildung 6.8: Mögliche Kollektion

Collection Members:	
4	signal
5	signal
6	signal
7	signal

Abbildung 6.9: Edit CollectionMembers

Alle Mitglieder einer Kollektion müssen dann dem Collection-Head hinzugefügt werden. Da das Datenmodell und die Datenbank immer nur einzelne verlinkte Ressourcen (T_LinkedResources) speichert, sind diese nur dem Kopf der Kollektion bekannt und können von den anderen Mitgliedern nicht gesehen werden [Abb.6.10].

Das würde bedeuten, dass nur der Kopf seinen nächsten und vorherigen Partner kennt, diese aber nicht deren Nachfolger (bzw. Vorgänger). So müsste bei jeder Ressource in der Kollektion alle Mitglieder einzeln eingetragen werden.

ResourceID	seeAlso	colletionMembers
4		4
4		5
4		6
4		7

Abbildung 6.10: CollectionHead

Da dies sehr aufwändig wäre, wurde mit einigen Funktionen Abhilfe geschaffen. Dazu wurde die Klasse RessourceData um einige Funktionalitäten erweitert.

```
267 public void setCollectionMembers(List<ResourceContainer> collectionMembers) {
268     this.collectionMembers = collectionMembers;
269     if (isHeadOfCollection()){
270         try {
271             setCollectionMembersToMembers(collectionMembers);
272         }catch (DBException e){
273             e.printStackTrace();
274         }
275     }
276 }
```

Abbildung 6.11: ResourceData setCollectionMembers()

Da die Methode ***setCollectionMembers()*** von jeder Ressource aufgerufen wird, spielt die Abbruchbedingung hier eine wichtige Rolle. Sonst würde die Methode ununterbrochen durch alle CollectionMembers springen und erst durch eine Exception gestoppt werden. Deshalb soll nur der CollectionHead die Methode ***setCollectionMembersToMembers()*** ausführen.

```
278 private void setCollectionMembersToMembers(List<ResourceContainer> collectionMembers){
279     List<Long> collectionMembersIDs = new ArrayList<>();
280     for (ResourceContainer rc : collectionMembers){
281         collectionMembersIDs.add(rc.getResource().getResourceID());
282     }
283     for (ResourceContainer rc : collectionMembers.subList(1,collectionMembers.size())){
284         if (!rc.getResource().isMemberOfCollection()){
285             for (Long id : collectionMembersIDs){
286                 T_LinkedResourcesDB4Servlet.createCollectionMembers
287                 (rc.getResource().getResourceID(),id);
288             }
289         }
290     }
291 }
```

Abbildung 6.12: ResourceData setCollectionMembersToMembers()

In dieser Methode werden zu Beginn alle Ressourcen-IDs in einer Liste gespeichert, da später die verlinkten Ressourcen nur aus einer Verbindung derer besteht.
In Zeile 283 wird dann durch die einzelnen CollectionMembers iteriert. Da die Mitglieder der Kollektion schon im Kopf gespeichert sind, beginnt diese List mit dem zweiten Element.
Die statische Methode ***createCollectionMembers(long, long)*** verbindet dann die aktuelle Ressourcen-ID mit der Liste der CollectionIDs und speichert diese unter T_LinkedResources in der Datenbank.
Um zu verhindern, dass eine Ressource mehreren Kollektion angehört, wird in Zeile 284 geprüft, ob diese bereits in einer Kollektion ist. Wäre eine Ressource in mehreren Kollektionen, wüsste die Logik nicht, in welcher Kollektion sich diese befindet.

Das Ergebnis des Beispiels in [Abb.6.8] würde folgende Spuren in der Datenbank hinterlassen:

ResourceID	seeAlso	colletionMembers
4		4
4		5
4		6
4		7
5		4
5		5
5		6
5		7
6		4
6		5
6		6
6		7
7		4
7		5
7		6
7		7

Abbildung 6.13: Datenbank T_LinkedResources

Jedes Mitglied der Kollektion kennt die gesamte Liste seiner Mitglieder. Somit ist sichergestellt, dass sich der Kopf nicht ändert und die Schleife über ihr Ziel hinausläuft.
Ist die Datenstruktur vorbereitet kann dann mithilfe der Methoden ***getPreviousCollectionMember()*** und ***getNextCollectionMember()*** durch die entstandene Kollektion gestöbert werden [Abb.6.3].

```
115 <!----------------CollectionMembers---------------------------------->
116<%
117 if (!resource.getCollectionMembers().isEmpty()) {
118 %>
119         <div class="slideshow-search">
120         <%=resource.getLinkPreviousCollectionMember() %>
121         <img src=<%=resource.getFirstContentImagePath()%> width="906"
122         <%=resource.getLinkNextCollectionMember() %>
123         </div>
124<%
```

Abbildung 6.14: CollectionMembers signaldetails.jsp

Da das Erstellen von mehreren Kollektionen dennoch einen erheblichen Aufwand darstellen würde, wurde eine zusätzliche Hilfsfunktion entwickelt. Diese Funktion ermöglicht das Zusammenstellen von Ressourcen mit **aufeinanderfolgenden IDs** in eine Kollektion. Dadurch kann in der Datenbank nach den IDs gesucht werden, um diese anschließend zusammenzuführen.

```
339 private static void setMultipleCollectionMembersInDirectIdOrder(int startID, int lastID){
340     Resource resource = ResourceProvider.getResourceById(startID).getResource();
341     List<ResourceContainer> collectionMembers = new ArrayList<>();
342     for (int i = startID ; i<=lastID ; i++){
343         collectionMembers.add(ResourceProvider.getResourceById(i));
344         T_LinkedResourcesDB4Servlet.createCollectionMembers(resource.getResourceID(),i);
345     }
346     resource.setCollectionMembers(collectionMembers);
347 }
```

Abbildung 6.15: setMultipleCollectionMembersInDirectIdOrder()

Dieser Methode werden sowohl die Start-ID als auch die letzte ID der gewünschten Kollektion übergeben. Der Ressourcenprovider greift mithilfe der Start-ID auf den Collection-Head zu. Daraufhin werden alle aufeinanderfolgenden Ressourcen in eine Liste aufgenommen und eine T_LinkedResource erstellt. Schließlich wird diese Liste dem Collection-Head hinzugefügt. Den Rest übernehmen dann die Funktionen, die zuvor beschrieben wurden, um diese Sammlung an alle Mitglieder weiterzuleiten [Abb.6.11] und [Abb.6.12].

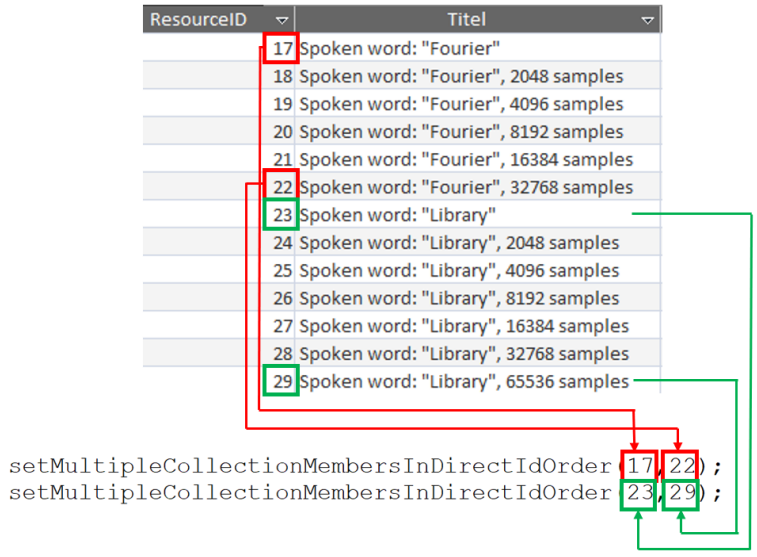

Abbildung 6.16: Bsp. setMultipleCollectionMembersInDirectIdOrder()

Kapitel 7

Migration bestehender Daten in neues Datenmodell

Da weder die Entwicklung des alten noch neuen Modells meine Hauptaufgabe gewesen ist, musste zuerst einige Zeit in das Verständnis der Datenmodelle investiert werden.
Dieses Kapitel lässt sich in zwei Abschnitte gliedern: Zum einen betrachten wir die Daten, die aus externen Datenbanken stammen, und zum anderen jene, die bereits im bestehenden Quellcode integriert wurden. Das neue Datenmodell ähnelte dem alten in großen Teilen, allerdings waren einige Anpassungen und Erweiterungen erforderlich.
Der Plan sah vor, dass durch die Anpassung der Import-Anweisungen eine nahtlose Verbindung zwischen dem alten und neuen Datenmodell hergestellt werden sollte. Dieses Vorhaben verlief größtenteils erfolgreich. In der Folge konnten veraltete Klassen ausgemustert werden. Aufgrund der Tatsache, dass insbesondere Bilder und Signale unmittelbar im Quellcode eingebettet waren, gestaltete sich die sofortige Löschung des Quellcodes als nicht umsetzbar.
Da sich nach diesem Vorgang alle Datenbestände in der neuen Datenbank befinden, ist der veraltete Code und der für die Migration später ebenfalls nicht auffindbar.

Die Hauptaufgabe für dieses Kapitel wurde von den Klassen ***InitResources*** und ***LiveDatabaseConnectionManager*** getragen.

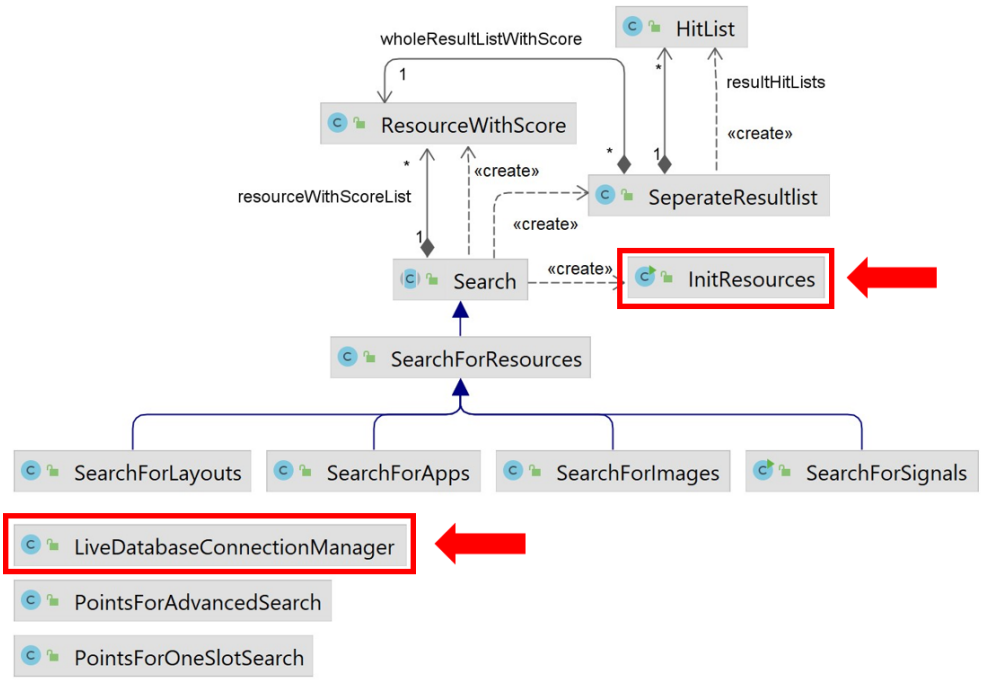

Abbildung 7.1: Übersicht Projekt, Migration

7.1 Migration aus bestehendem Quellcode

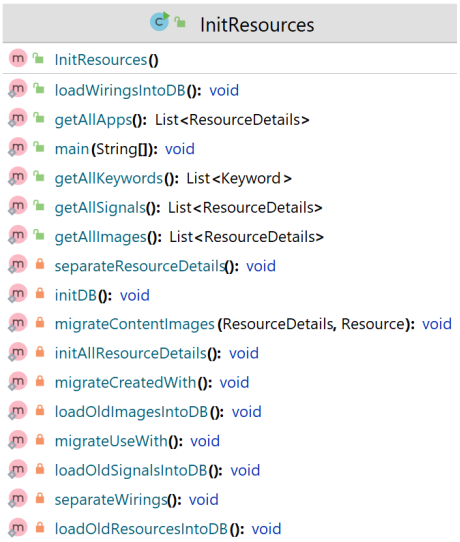

Abbildung 7.2: Klasse InitResources

Während des Datenmigrationsprozesses aus dem Quellcode werden zunächst die alten Datensätze mithilfe der Methode ***initAllResourceDetails()*** in eine Liste überführt. Gleichzeitig werden die zugehörigen Handler für Schlüsselwörter, Formeln, Classifier und Bilder initialisiert, um eine reibungslose Verarbeitung sicherzustellen. Darauf aufbauend ermöglicht die Methode ***initResource.getList()*** den Zugriff auf die Liste aller verfügbaren Ressourcen. Anschließend erfolgt die Aufteilung dieser Liste mithilfe der Funktion ***separateResourceDetails()*** in separate Gruppen, nämlich Signale, Bilder und Apps. Dies ist vonnöten, da jede dieser Ressourcen einzeln in die Datenbank integriert werden muss, um einen sicheren Ablauf zu gewährleisten. Im Weiteren konzentrieren wir uns auf das Beispiel der Signale: Zuerst wird eine Schleife durch die bereits sortierten Signale durchlaufen. Dabei wird jede Ressource in ein Signal umgewandelt, um an alle Parameter zu gelangen.

```
167  private static void loadOldSignalsIntoDB() throws DBException {
168      for (ResourceDetails resourceDetails : allSignals) {
169          SignalDetails signalDetails = (SignalDetails) resourceDetails;
170          Signal signal = signalDetails.getSignal();
171          SignalResource dbSignal = new SignalResource();
```

Abbildung 7.3: InitResources loadOldSignalsIntoDB() Teil1

```
182 String title = signalDetails.getTitle();
183 String description = signalDetails.getDescription();
184
185 List<Keyword> keywordList = signalDetails.getKeywords();
186
187 //SignalDetails
188 double power = signal.getPower();
189 double peakValue = signal.getPeakValue();
190 long samples = signal.getNumberOfSamples();
191 List<Classifier> classifierList = signal.getClassifier();
192 Format format = signal.getFormat();
193 double length = signal.getLengthInSec();
194 double sampleRate = signal.getSampleRate();
195 FileSize fileSize = signal.getFileSize();
```

Abbildung 7.4: InitResources loadOldSignalsIntoDB() Teil2

Nun erfolgt die Extraktion sämtlicher existierender Daten aus der vorherigen Ressource, welche vorerst in temporären Variablen zwischengespeichert werden. Ein direktes Schreiben in die neue Ressource ist nicht unmittelbar möglich, da einige Parameter möglicherweise nicht ausgefüllt sind und dies zuvor zu einer Ausnahme führen könnte. Daher werden solche potenziellen Ausnahmen präventiv abgefangen. Sobald die Informationen erfolgreich aufbereitet sind, können sie in die neue Ressource übertragen werden. Für einige neu hinzugekommene Parameter, die im alten Datenmodell nicht vorhanden waren, bedarf es eine manuelle Zuweisung entsprechender Parameterwerte[Abb.7.5]:

Für den Parameter *SaveDate* wird das Datum der letzten Migration gespeichert. Der *PublishedStatus* muss die drei besitzen, da dies bedeutet die Ressource ist öffentlich, was sie zuvor auch war.

```
213 dbSignal.setSaveDate(LocalDate.now().toString());
214 //PublishedNumber
215 dbSignal.setPublishedStatus(3);
216 //userID
217 dbSignal.setUserID(userIdForTest);
218 //contentPath
219 dbSignal.setContentPath(contentPath);
```

Abbildung 7.5: InitResources loadOldSignalsIntoDB() Teil3

Die *UserID* wurde für Testzwecke auf die eins gesetzt, was für den ersten User in der Datenbank steht. Der *ContentPath* wurde der Datenbankstruktur neu hinzugefügt. Dieser ist die Verlinkung zum Download eines Signals.

```
226 //create Signal in DB
227 SignalDB4Servlet.createSignal(dbSignal);
228
229 //contentImage
230 migrateContentImages(signalDetails, dbSignal);
231 //keywords
232 try {
233     for (Keyword k : keywordList) {
234         k.setResourceID(dbSignal.getResourceID());
235         T_KeywordDB4Servlet.createTKeyword(k);
236     }
237 } catch (NullPointerException e) {
238     System.out.println(dbSignal.getTitel() + " has no Keywords");
239 }
```

Abbildung 7.6: InitResources loadOldSignalsIntoDB() Teil4

In Zeile 227 erfolgt die Eintragung des neu generierten Signals in die Datenbank. Dieser Schritt ist von essenzieller Bedeutung und sollte vor den abschließenden beiden Aktionen durchgeführt werden. Um die Verknüpfungen mit Schlüsselwörtern und Bildern zu realisieren, ist es unerlässlich, dass die neue Ressource bereits eine eindeutige ID besitzt. So kann gewährleistet werden, dass die notwendigen Verlinkungen korrekt erfolgen. Sofern die Ressource nicht zuvor in die Datenbank eingetragen wurde, fehlt ihr die besagte ID, und in der Folge ist es auch nicht möglich, sie mit Schlüsselwörtern oder Bildern zu verknüpfen.
Die Methode ***migrateContentImages()*** realisiert ihre Funktion, indem sie die Bestandteile eines Pfades in der neuen Struktur gezielt anpasst. Dieser Anpassungsprozess ist notwendig, da im aktualisierten Datenmodell die Parameter andere Bezeichnungen tragen und aus verschiedenen Elementen zusammengesetzt sind. Es gilt, die Struktur der Methode entsprechend anzupassen, um eine reibungslose Transformation der Pfade zu gewährleisten. Dieses Vorgehen ist essenziell, um sicherzustellen, dass die Pfadangaben korrekt im Kontext des neuen Datenmodells funktionieren [Abb.A.10].

Die Vorgehensweise der Migration von Bildern gleicht weitgehend derjenigen für Signale, mit Ausnahme einiger differenzierter Parameter.

7.2 Migration aus bestehenden Datenbanken

Ursprünglich war keine eigenständige Klasse für die Datenbankmigration vorgesehen, da anfänglich nur eine Datenbank involviert war. Mit der fortschreitenden Migration von insgesamt drei unterschiedlichen Datenbanken wurde jedoch die Notwendigkeit offensichtlich, eine dedizierte Klasse für diese Aufgabe zu erstellen.

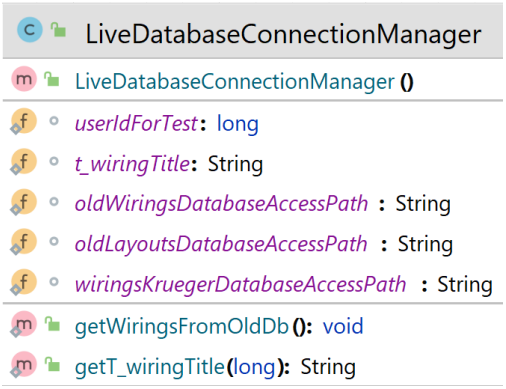

Abbildung 7.7: Klasse LiveDatabase-ConnectionManager

Aufgrund der begrenzten Zeit, die zur Verfügung stand, um mich mit Datenbanksprachen und Verbindungsmechanismen vertraut zu machen, konnte ich durch die Bachelorarbeiten meiner Kommilitonen, auf bewährte Muster zurückgreifen.
Ebenfalls an dieser Stelle liegt der Fokus auf der Beschreibung der Datenbankmigration für bestehende Wirings, da der Ablauf für Layouts und andere Wirings identisch ist. Besondere Sorgfalt ist hierbei geboten, da die Unterschiede in Feldgrößen zwischen der alten und der neuen Datenbank exakt berücksichtigt werden müssen. Eine ungenaue Abstimmung kann zu langwierigen Fehlerbehebungsprozessen führen, da in der Regel keine explizite Fehlermeldung ausgegeben wird.
Ein weiteres Augenmerk liegt auf der korrekten Übernahme der einzelnen Spalten oder Tabellen aus den jeweiligen Datenbanken. Hierbei ist entweder eine exakte Kopie oder äußerste Sorgfalt bei der manuellen Übertragung notwendig, um potenzielle Fehler von vornherein zu vermeiden. Die präzise Handhabung dieser Aspekte trägt dazu bei, Unstimmigkeiten zu minimieren und einen sauberen Migrationsprozess zu gewährleisten.

```
26  public static void getWiringsFromOldDb(String dbAccessPAth) throws SQLException{
27      //Get connection to database
28      Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
29      Connection connection = DriverManager.getConnection("jdbc:ucanaccess://"
30                                                          + dbAccessPAth);
31      String selectString = "SELECT * FROM T_Wiring";
```

Abbildung 7.8: LiveDatabaseManager getWiringsFromOldDb() Teil1

Die Methode ***getWiringsFromOldDb()*** erhält als Parameter den Pfad zur Datenbank. In der Abfolge wird der Treiber für die Datenbank initialisiert. Anschließend wird mithilfe dieses Treibers und dem Pfad eine Datenbankverbindung hergestellt (siehe Zeile 29). Wie bereits auf der vorherigen Seite erläutert, ist der richtige Name der gewünschten Tabelle aus der Datenbank in Zeile 31 von besonderer Relevanz(grün).

```
35  DBSelectCommand cmd = new DBSelectCommand(connection, selectString, null) {
36      WiringResource wiringResource = new WiringResource();
37      @Override
38      protected Object handleResultSet(ResultSet rs) throws Exception {
39          while (rs.next()) {
40              wiringResource.setResourceType("wiring");
41              wiringResource.setTitel(rs.getString("Title"));
42              wiringResource.setDescription(rs.getString("WiringDefinition"));
43              wiringResource.setSaveDate(LocalDate.now().toString());
44              wiringResource.setPublishedStatus(3);
45              wiringResource.setUserID(userIdForTest);
46              wiringResource.setTitle(rs.getString("Title"));
47              wiringResource.setClassname(rs.getString("Wiring"));
48              wiringResource.setWiringDefinition(rs.getString("WiringDefinition"));
49              wiringResource.setUserChanges(rs.getString("UserChanges"));
50              wiringResource.setVersion(rs.getInt("Version"));
51              try {
52                  WiringDB4Servlet.createWiring(wiringResource);
53              } catch (NullPointerException e) {
54                  e.printStackTrace();
55              }
```

Abbildung 7.9: LiveDatabaseManager getWiringsFromOldDb() Teil2

Im weiteren Verlauf der Funktion (Zeile 36) erfolgt die Erzeugung einer neuen Ressource "WiringResource". Dann durchläuft das ResultSet (rs) die angegebene Datenbanktabelle und überträgt die jeweiligen Spaltenwerte in die frisch erstellte Ressource. Ein besonderes Augenmerk sollte erneut auf die Bedeutung der Spaltennamen gelegt werden(rot). Ihre exakte Zuordnung ist von zentraler Bedeutung für den korrekten Ablauf. Ebenfalls ist es von hoher Wichtigkeit, die Spaltenbezeichnung beider Datenbanken genau zu analysieren. Nur so ist gewährleistet, dass die Zuordnung der Spalten zu den jeweiligen Ressourcenparameter korrekt erfolgt, denn oft weichen die Bezeichnungen dieser Spalten zwischen den Datenbanken voneinander ab.

Sobald sämtliche Parameter durchlaufen sind, erfolgt in Zeile 52 die Einspeisung dieser Ressource samt ihrer Parameter in die neue Datenbank.

7.3 Migration createdWith und useWith

Die Migration der Verknüpfungen zwischen Signalen und Bildern zusammen mit den Wirings stellte zweifellos eine erhebliche Herausforderung dar. Dies lag daran, dass die Wirings in der Datenbank gespeichert waren, während die Verbindungen zu diesen in den Quellcode eingebettet sind. Innerhalb des Quellcodes wurden die Wirings allerdings lediglich durch eine eindeutige ID ohne Titel identifiziert. Dies erforderte eine beträchtliche Anzahl an Vergleichen und Abfragen, sowohl auf der Seite der Datenbank als auch innerhalb des Quellcodes selbst.

```
354* private static void migrateCreatedWith() throws DBException, ClassNotFoundException, SQLException {
355      for (Resource image : allDbImages) {
356          String dbTitle = image.getTitel();
357          for (ResourceDetails rd : allImages) {
358              String detailsTitle = rd.getTitle();
359              if (dbTitle.equals(detailsTitle)) {
360                  for (ResourceDetails cw : rd.getCreatedWith()) {
361                      if (cw instanceof MyWiringAppDetails) {
362                          String t_wiringTitle = LiveDatabaseConnectionManager.getT_wiringTitle(
363                                  ((MyWiringAppDetails) cw).getMyWiringId());
364                          for (Resource wr : allDbWirings) {
365                              String r_wiringTitle = wr.getTitel();
366                              if (t_wiringTitle.equals(r_wiringTitle)) {
367                                  T_LinkedResourcesDB4Servlet.createCreatedWith(image.getResourceID(),
368                                          wr.getResourceID());
```

Abbildung 7.10: InitResources migrateCreatedWith()

Die Methode startet, indem sie zunächst alle Bilder in der Datenbank durchläuft und ihre Titel speichert. Anschließend erfolgt dasselbe Verfahren mit den Bildern im Quellcode. Falls die Titel übereinstimmen kann der Prozess fortgesetzt werden. Von dieser Ressource aus werden alle zugehörigen Ressourcen mit der „createdWith“-Verknüpfung in Zeile 360 durchgegangen. Falls diese Ressourcen Instanzen von „MyWiringAppDetails“ sind, handelt es sich um die richtigen Objekte.
In Zeile 362 und 363 wird die Methode ***getT_WiringTitel()*** [Abb.A.11] verwendet, um den Titel anhand der Wiring-ID aus der T_Wiring-Datenbank abzurufen. Danach wird eine Iteration durch alle Wirings in der neuen Datenbank durchgeführt.
Der Titel jedes Wirings wird ebenfalls gespeichert und mit dem Titel aus der T_Wiring Datenbank verglichen. Wenn die Titel übereinstimmen, werden die Verknüpfungen in der neuen Datenbank erkannt.

Sobald diese Übereinstimmung festgestellt ist, erfolgt mithilfe der Funktion ***T_LinkedResourcesDB4Servlet.createCreatedWith(long, long)*** die Erstellung der Verknüpfung unter Verwendung ihrer IDs in der neuen Datenbank.

1. Titelvergleich Quellcode und neue Datenbank (Z.359)
2. CreatedWith-ID und Titel aus T_Wiring Datenbank (Z. 360-363)
3. Titelvergleich T_Wiring und R_Wiring (Z.366)
4. T_LinkedResources in neuer Datenbank mit IDs (Z.367-368)

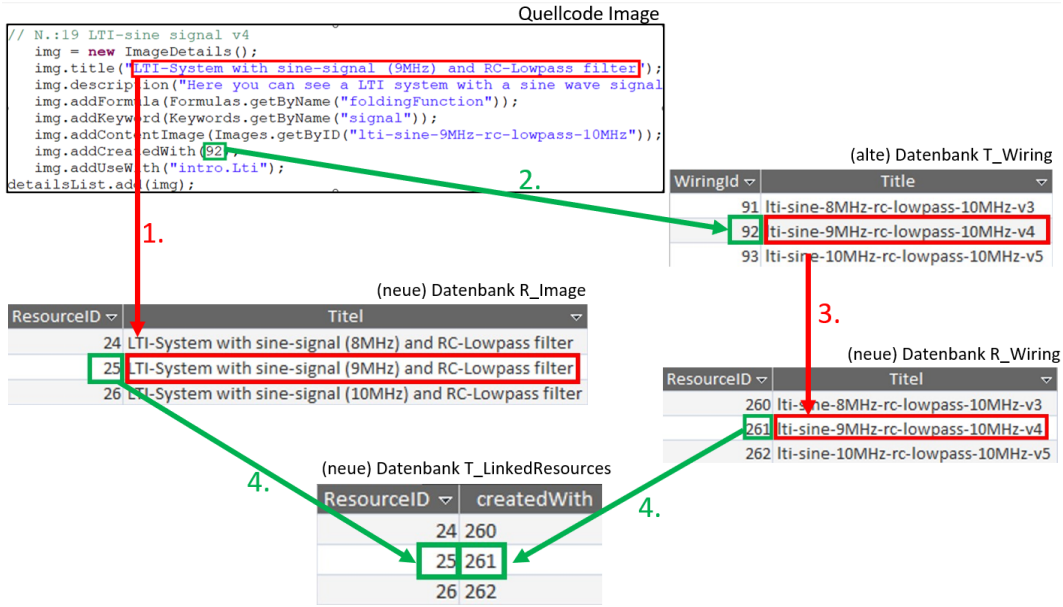

Abbildung 7.11: Vorgehensweise createdWith-Resources

Ein weiterer wichtiger Aspekt bei der Migration der bestehenden Ressourcen betrifft deren Reihenfolge. Es ist von entscheidender Bedeutung zu beachten, dass zunächst alle Ressourcen in die Datenbank eingetragen werden müssen, bevor die Verknüpfungen zwischen ihnen hergestellt werden kann. Diese Vorgehensweise hat ihren Grund darin, dass einige Zeit benötigt wird, bis sämtliche Ressourcen erfolgreich in die Datenbank geschrieben werden.

Wird das Programm in direkter Abfolge ausgeführt, finden die Verknüpfungsmethoden keine entsprechenden Ressourcen in der Datenbank. Dies hat zur Folge, dass keine Verknüpfungen erstellt werden können.
Die Verknüpfungen der „useWith“-Ressourcen folgen demselben Prinzip. Allerdings werden in den Quellcode-Ressourcen keine IDs angegeben, sondern der Wiring-Classname. Dadurch erfolgt der letzte Vergleich anhand diesem Wiring-Classname [Abb.A.12].

Kapitel 8

Hitlists

Die Erstellung der Hitlisten trägt zweifelsfrei zur erheblichen Steigerung der Benutzerfreundlichkeit und zur Beschleunigung der Suchprozesse bei. Dabei sind sie in ihrer Umsetzung und Darbietung gezielt auf die Relevanz der einzelnen Suchergebnisse abgestimmt. Das übergeordnete Ziel ist, eine rasche Übersicht der erzielten Suchergebnisse zu bieten und den Nutzern einen unmittelbaren Einblick in die verschiedenen Bereiche dieser Ergebnisse zu gewähren. Dieses Vorgehen trägt dazu bei, die Effizienz der Informationsgewinnung zu erhöhen und die Nutzererfahrung insgesamt zu optimieren.
Im gesamten Projekt verwendet dieser Teil die Klassen ***Hitlist***, ***SeperateResultlist*** und ***ResourceWithScore***.

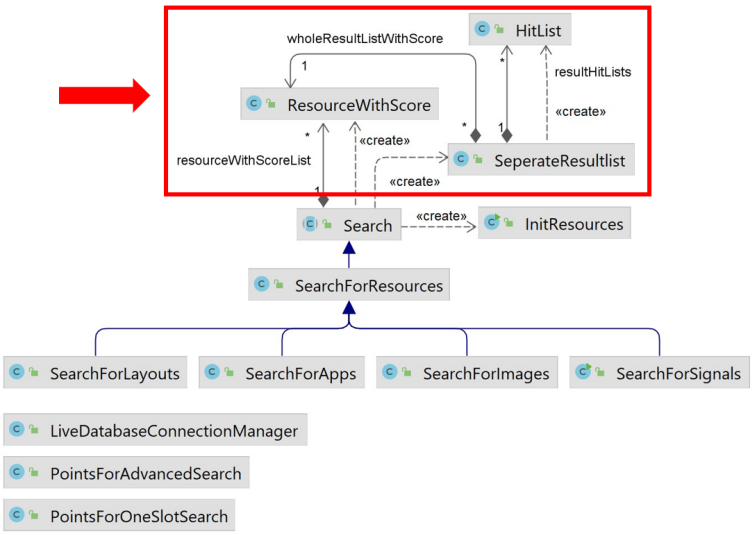

Abbildung 8.1: Übersicht Projekt, Hitlists

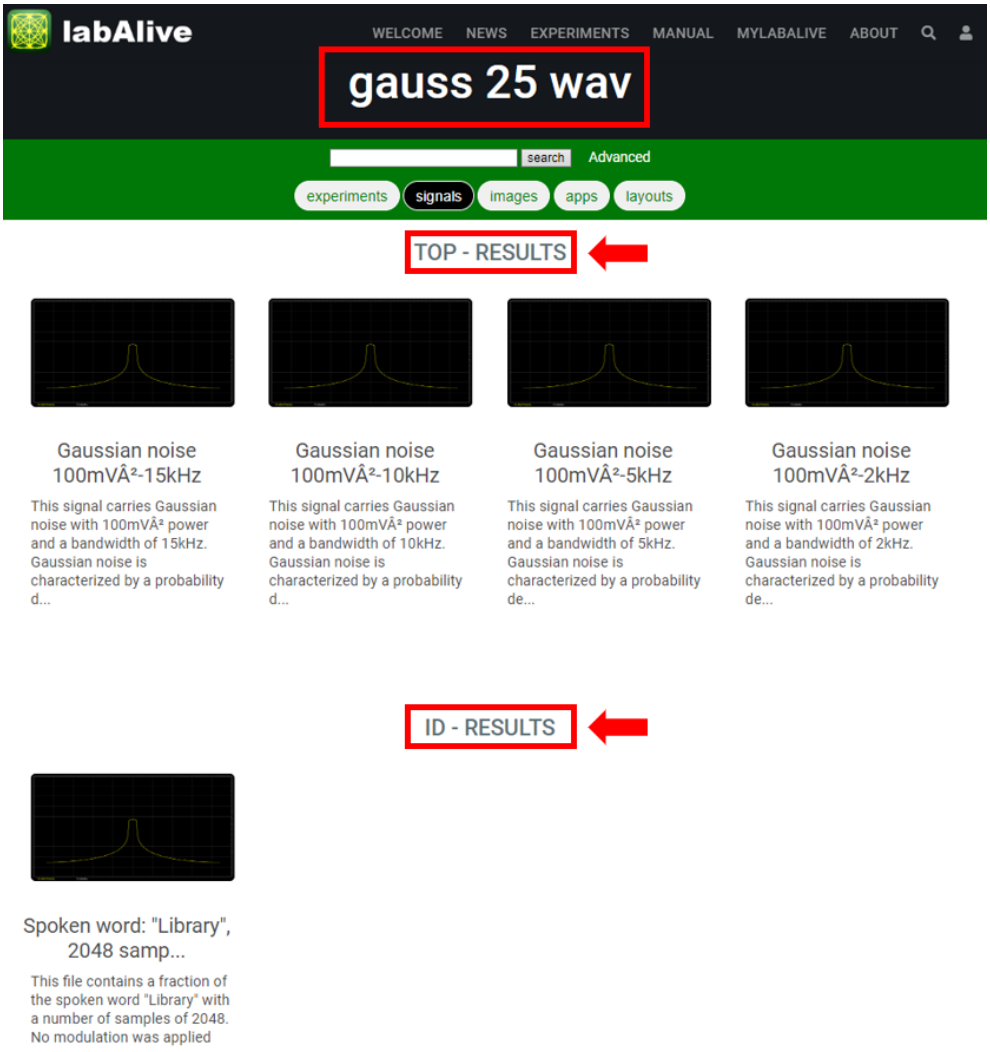

Abbildung 8.2: Ansicht Suchergebnisse

Die grundlegende Idee hinter diesem Ansatz besteht darin, die Suchergebnisliste basierend auf den verschiedenen Parametern aufzuteilen und in individuellen Hitlisten zu präsentieren. Dadurch erhält der Nutzer auf einen Blick eine klare Übersicht darüber, wo die Ergebnisse erzielt wurden, ohne sich durch eine mühsame Durchsicht aller Ergebnisse klicken zu müssen. Diese Segmentierung ermöglicht es auch, die Reihenfolge der einzelnen Listen festzulegen und eine geordnete Darstellung zu gewährleisten. Die erste Liste, die TOP-ResultList, setzt sich stets aus den Ressourcen mit den meisten Trefferpunkten zusammen. Dies gibt dem Administrator die Möglichkeit zu bestimmen, aus wie vielen Ressourcen diese Liste bestehen soll.

Anschließend folgt das Ergebnis der ID-Suche, welches natürlich nur aus einer einzigen Ressource bestehen darf, da pro ID nur eine Ressource existiert. Daraufhin werden Titel, Beschreibung und die weiteren Parameter aufgelistet. Dieser Aufbau erleichtert es Nutzern, relevante Informationen schnell zu erfassen und stellt sicher, dass die Bedeutung der Ergebnisse klar und effizient kommuniziert wird.
Sollte kein Ergebnis gefunden werden, wird dem Nutzer angezeigt, dass die Ergebnisliste leer ist, und es werden alle verfügbaren Ressourcen angezeigt.
Eine weitere Verbesserung der Übersichtlichkeit besteht darin, dass bei Ressourcen, die Teil einer Kollektion sind, lediglich der Kopf der Kollektion angezeigt wird. Durch einen Klick auf diesen Kopf, wie in [Abb.6.3] beschrieben, kann der Nutzer die Kollektion erkunden.
Wenn jedoch ein Suchergebnis vorliegt, werden die einzelnen Ressourcen einer Kollektion separat angezeigt. Dies ist darauf zurückzuführen, dass gezielt nach individuellen Daten gesucht wird, die dementsprechend detailliert angezeigt werden sollen. Bei der Übersichtsansicht hingegen werden nur Kollektionen angezeigt, um die Übersichtlichkeit zu wahren.

8.1 Class HitList

Die Klasse ***HitList*** stellt mit ihren Parametern und Methoden den Grundbaustein dieser Funktionalität. Eine HitList hat einen Namen als String und eine Liste von Ressourcen, die als mögliche Ergebnisse dienen. Es ist möglich, sowohl einzelne Elemente als auch ganze Listen von Ressourcen hinzuzufügen, zum Beispiel für die Top-Liste, bei der ein kleiner Teil der gesamten Liste abgeschnitten wird.

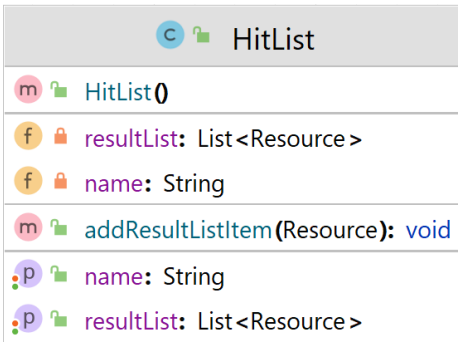

Abbildung 8.3: Klasse HitList

8.2 Class ResourceWithScore

Die Klasse ***ResourceWithScore*** hat in ihrer grundlegenden Funktionalität keine wesentlichen Änderungen gegenüber ihrer ursprünglichen Implementierung. Allerdings wurden ihr viele zusätzliche Parameter hinzugefügt, welche rechts dargestellt sind. Für jeden Parameter gibt es nun einen eigenen Integer. Jede Ressource wird dann mit diesen Parametern verknüpft. Außerdem existiert ein Parameter, der die Gesamtzahl der Trefferpunkte repräsentiert, die sich aus der Summe der einzelnen Zahlen zusammensetzt. Zusätzlich implementiert die Klasse das Comparable-Interface, was bedeutet, dass sie nach ihrem Wert sortiert werden kann.
Sollten zukünftig noch weitere Ressourcen oder Parameter hinzugefügt werden, so können diese hier einfach erweitert werden.

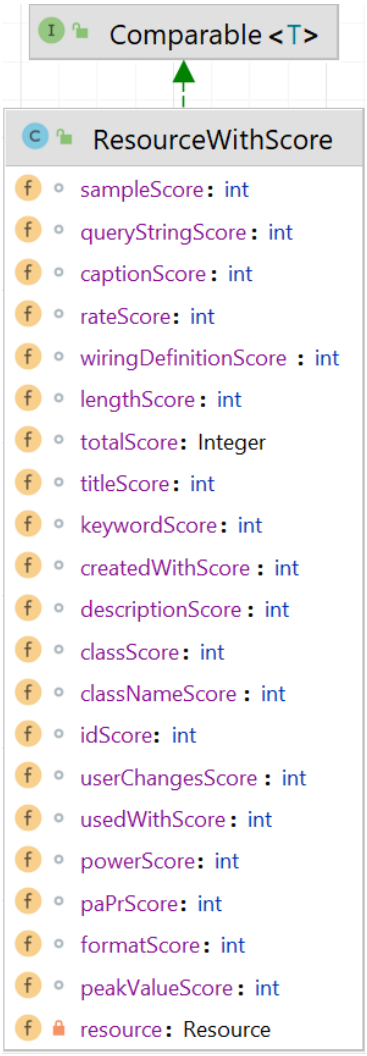

Abbildung 8.4: Klasse ResourceWithScore

8.3 Class SeperateResultlist

Die Klasse ***SeperateResultlist*** empfängt von der Suchmaschine die vollständige Liste der gefundenen Ressourcen zusammen mit ihren Bewertungen. Die Hauptaufgabe dieser Klasse besteht darin, diese Ressourcen in die entsprechenden Listen aufzuteilen.

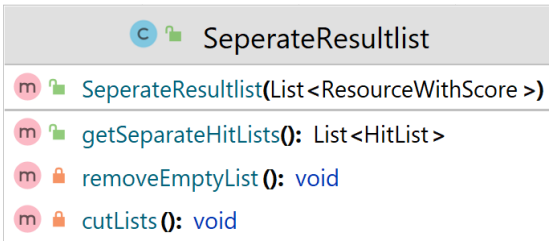

Abbildung 8.5: Klasse SeperateResultlist

Dies geschieht, indem jeder Bewertungspunkt überprüft wird und die Ressource dann der zugehörigen Liste zugewiesen wird.

```
 9 public class SeperateResultlist {
10     static final int lengthOfTopHitlist = 4;
11     static final int lengthOfOtherHitlists = 4;
```

Abbildung 8.6: Hitlist-Listenlänge

Wie zuvor erwähnt, hat der Administrator die Möglichkeit, die Länge der Listen einzustellen. Gegenwärtig bestehen alle Listen aus vier Elementen. In dieser Klasse werden zu Beginn alle möglichen Listen für die individuellen Parameter instanziiert.
Die Suchergebnisliste der Suchmaschine wird bereits in sortierter Reihenfolge an diese Klasse übergeben. Daher muss für die "Top-resultlist"nur der Anfangsteil bis zur gewünschten Länge (*lengthOfTopHitlist*) abgeschnitten werden. Der verbleibende Teil dieser Liste wird dann in ein Sortiersystem eingebracht und den jeweiligen Listen zugeordnet.
Da nicht für jede Ressource eine separate Liste erstellt werden soll, werden am Ende alle Listen hinzugefügt und anschließend die leeren Listen entfernt. Danach werden diese Listen ebenfalls auf die gewünschte Länge zugeschnitten. Sobald dieser Prozess abgeschlossen ist, werden diese Listen der endgültigen Ergebnisliste (*finalResultHitLists*) hinzugefügt und an die jeweilige Anzeigeseite übergeben.

Wenn, wie bereits beschrieben, kein Suchergebnis vorliegt, kommt die Methode ***removeCollectionMembers*** aus der übergeordneten Klasse ***Search*** zum Einsatz. Diese Methode durchsucht alle Ressourcen der entsprechenden Art und entfernt dabei die Elemente, die Teil von Sammlungen sind. Die bereinigte Liste wird anschließend auf der Ergebnisseite angezeigt.

```
156 private List<Resource> removeCollectionMembers(List<Resource> resourceList){
157     List<Resource> resourcesWithoutCollectionMembers = new ArrayList<>();
158     for (Resource resource : resourceList){
159         if (!resource.isMemberOfCollection()){
160             resourcesWithoutCollectionMembers.add(resource);
161         }
162     }
163     return resourcesWithoutCollectionMembers;
164 }
```

Abbildung 8.7: Search removeCollectionMembers()

Kapitel 9

Fazit

Abschließend lässt sich festhalten, dass der optimale Suchalgorithmus eine entscheidende Rolle bei der Effizienz und Genauigkeit von Suchprozessen auf Webseiten und in Datenbanken spielt. Ein gut gestalteter Suchalgorithmus berücksichtigt verschiedene Faktoren wie Suchbegriffe, Ranking-Methoden und Filtermechanismen, um den Benutzern relevante Ergebnisse schnell und präzise zu präsentieren.
Die Benutzerfreundlichkeit auf Webseiten ist von großer Bedeutung, um eine angenehme und effektive Nutzungserfahrung zu gewährleisten. Eine intuitive Benutzeroberfläche, klare Navigation und ansprechendes Design tragen dazu bei, dass Nutzer sich leicht auf der Seite zurechtfinden können. Ein guter Suchalgorithmus unterstützt diese Benutzerfreundlichkeit, indem er schnell relevante Ergebnisse liefert und dem Nutzer ermöglicht, präzise zu filtern und die gewünschten Informationen zu finden.
Insgesamt ist eine gelungene Kombination aus einem leistungsfähigen Suchalgorithmus, einer benutzerfreundlichen Website und einer gut gestalteten Datenbank entscheidend, um den Nutzern eine reibungslose und effiziente Erfahrung bei der Suche nach Informationen zu bieten. Die stetige Weiterentwicklung dieser Aspekte trägt dazu bei, die Qualität der Online-Interaktionen zu verbessern und die Anforderungen der Benutzer zufriedenstellend zu erfüllen.

Kapitel 10

Ausblick

Der unaufhaltsame technische Fortschritt wird sich auch in Zukunft fortsetzen. Daher ist eine kontinuierliche Beobachtung und Weiterentwicklung von enormer Bedeutung, um nicht den Anschluss zu verlieren. In Anbetracht der Tatsache, dass Zeit einen maßgeblichen Faktor in dieser Arbeit darstellt, ergeben sich noch einige Bereiche, in denen Verbesserungen oder Weiterentwicklungen möglich sind:

1. Verbesserte Suchanfragen-Analyse: Durch die Verwendung eines präzisen Suchalgorithmus können Website-Betreiber wertvolle Einblicke in das Verhalten der Benutzer gewinnen.
2. Bei der Erstellung von HitLists (Kap. 8) besteht die Möglichkeit, den jeweiligen Score in eine eigene Klasse einzubinden, um eine separate Sortierung desselben zu ermöglichen.
3. Der Ort für die Aktualisierung des „published“-Status sowie für die Aktualisierung der Collection-Members muss noch geplant werden. Eine mögliche Lösung könnte darin bestehen, dies durch eine Aktualisierung im Cache zu bewerkstelligen, da die Daten aus verschiedenen Ebenen der Datenbank abgerufen werden und ansonsten ein Neustart des Servers erforderlich wäre.
4. Fehlersuche: Bei Aktualisierung einer Ressource geht ihr „published“-Status verloren.
5. Gestaltung und Darstellung: Aufgrund der begrenzten Zeit, die für die Auseinandersetzung mit HTML und CSS zur Verfügung stand, können einige Designs noch überarbeitet werden.

6. Wie in [Abb.6.13] erklärt wurde, muss jede Ressource in der Verknüpfung mit der gesamten Liste einzeln verlinkt sein. In Zukunft könnten erhebliche Datenbankeinträge eingespart werden, wenn es möglich wäre, mehrere IDs in eine verknüpfte Ressource einzutragen.
Zum Beispiel:

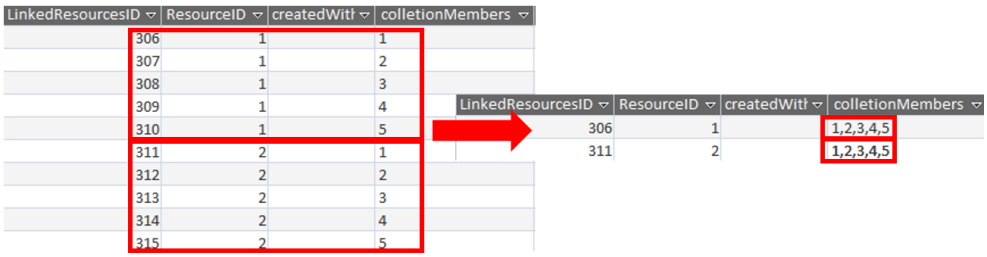

Abbildung 10.1: Verbesserungsvorschlag T_LinkedResources

Anhang A

Anhang

Im Anhang befinden sich einige Methoden oder Klassendiagramme. Da an diesem Projekt ununterbrochen weitergearbeitet wird, können diese bereits leicht abgeändert sein.

```
67  void removeResourceContainer(List<ResourceContainer> resourceContainers,
68          List<Resource> resourceList) {
69      for (ResourceContainer rc : resourceContainers) {
70          resourceList.add(rc.getResource());
71      }
72  }
```

Abbildung A.1: Class Search removeResourceContainer()

```
173 public List<HitList> getHitlists() {
174     Collections.sort(resultListWithScore);    // ascending order
175     Collections.reverse(resultListWithScore); // descending order
176     SeperateResultlist seperateResultlist = new SeperateResultlist(resultListWithScore);
177     return seperateResultlist.getSeparateHitLists();
178 }
```

Abbildung A.2: Class Search getHitlists()

```
151 public HitList getAllResources() {
152     HitList ret = new HitList();
153     ret.setName("");
154     ret.setResultList(removeCollectionMembers(resourceList));
155     return ret;
156 }
```

Abbildung A.3: Class search getAllResource()

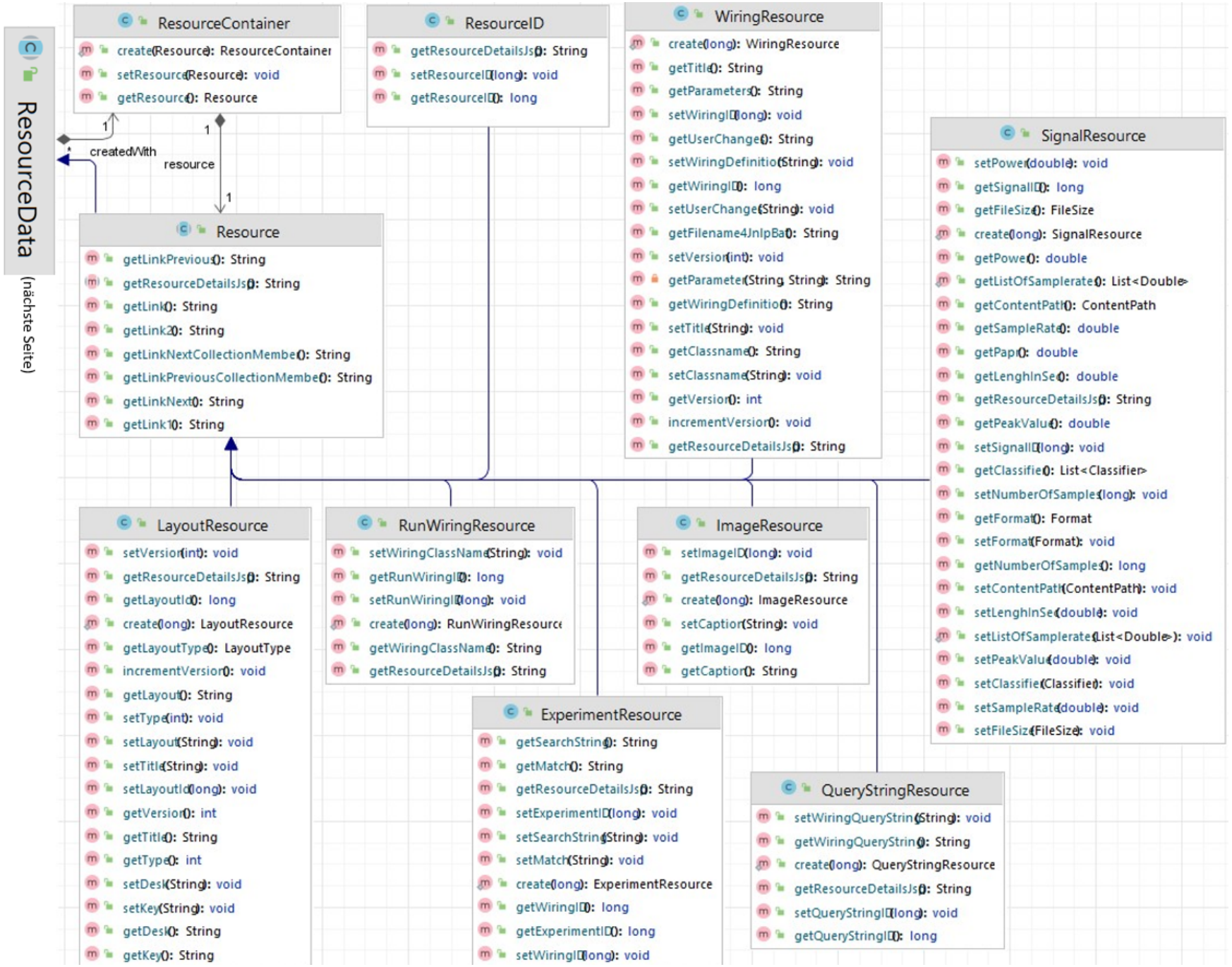

Abbildung A.4: package resource.data

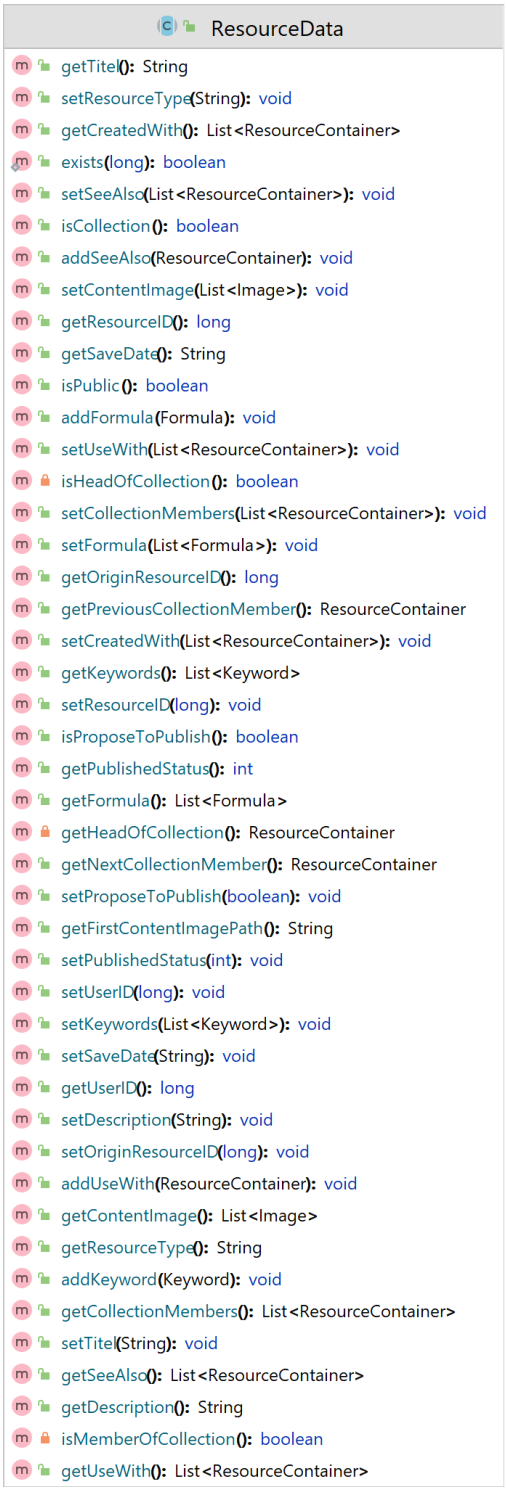

Abbildung A.5: Class ResourceData

Abbildung A.6: package resource.data.additional

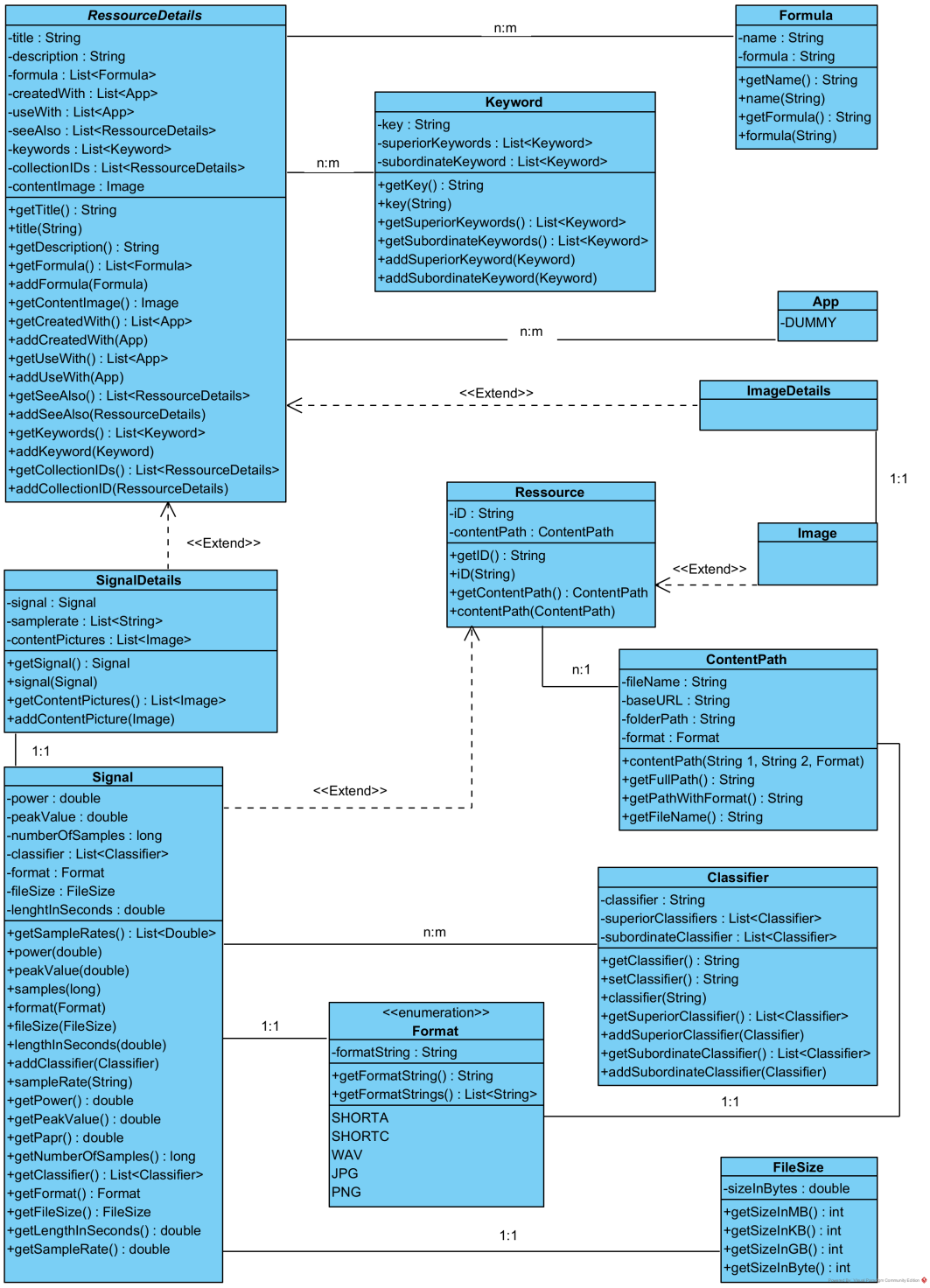

Abbildung A.7: Altes Datenmodell

```
14 long id = Long.parseLong(request.getParameter("id"));
15 Resource resource = ResourceProvider.getResourceById(id).getResource();
16 String uuid = Session.instance(request).getUuid();
17 String titel = resource.getTitel();
18
19 if(!(resource.isPublic() || resource.isUsersResource(request))){
20     %>
21     <jsp:include page="/jsp/header.jsp" />
22     <section id="home_experiment">
23         <hr>
24         <h1>This Resource is not public or not yours. Logged in?</h1>
25     </section>
26     <%
27 }else{
28 <!-------------------Head---------------------------------->
29 %>
30 <jsp:include page="/jsp/header.jsp" />
31
32 <section id="home_experiment">
33     <hr>
34     <h1><%=  titel %></h1>
35
36 </section>
37 <section id="content">
38 <p><%=  resource.getDescription() %></p>
39
40 <%
41 if(!resource.getFormula().isEmpty()){
42     for(Formula formel : resource.getFormula()){
43         if(formel != null){%>
44
45             <h2><%="Formula: " + formel.getFormula()%></h2>
46
47     <%}}
48 }%>
49 <h2><%= "ClearTitle: " + titel%></h2>
50 <!-------------------Head END---------------------------------->
51
52 <!-------------------Signal Elements------------------------->
53 <% try{
54     SignalResource sr = (SignalResource) resource;
55     %>
56     <table>
57         <tr>
58             <th style="text-align:right;">Power:</th>
59             <td style="width:65%;"><%=sr.getPower()%></td>
60         </tr>
61         <tr>
62             <th style="text-align:right">Peak-Value:</th>
63             <td><%=sr.getPeakValue()%></td>
64         </tr>
65         <tr>
66             <th style="text-align:right">PAPR:</th>
67             <td><%=sr.getPapr()%></td>
68         </tr>
69         <tr>
70             <th style="text-align:right">Number of Samples:</th>
71             <td><%=sr.getNumberOfSamples()%></td>
72         </tr>
73         <tr>
74             <th style="text-align:right">Samplerate:</th>
75             <td><%=sr.getSampleRate()%></td>
76         </tr>
77     </table>
78     <%
79    }catch (ClassCastException e){
80     e.printStackTrace();
81    }
82 %>
83 <!-------------------Signal Elements END---------------------->
```

Abbildung A.8: signaldetails.jsp Teil 1

```
<!------------------Download------------------------------->
<%
try{
    SignalResource sr = (SignalResource) resource;
    %>
    <p><a href= <%=sr.getContentPath().getPathWithFormat()%> >Download Signal </a></p>
    <%
   }catch (ClassCastException e){
    e.printStackTrace();
   }
%>
<!------------------ContentImages------------------------------->
<%
if(!resource.getContentImage().isEmpty() && !resource.isCollection()){

for(Image img : resource.getContentImage()){
    %>
    <img src=<%=img.getWholeContentImagePath()%> width="906" />
    <%
}}
%>
<!----------------CollectionMembers---------------------------------->
<%
if (!resource.getCollectionMembers().isEmpty()) {
%>
        <div class="slideshow-search">
        <%=resource.getLinkPreviousCollectionMember() %>
        <img src=<%=resource.getFirstContentImagePath()%> width="906" height="500" />
        <%=resource.getLinkNextCollectionMember() %>
        </div>
<%
}
%>
<br>
<br>
<!----------------createWith---------------------------------->
<br>
<%
if(!resource.getCreatedWith().isEmpty()){
%>
    <p><b>This image has been created using this labAlive app:</b></p>
<%
    for (ResourceContainer createdWith : resource.getCreatedWith()) {
    %>
    <div style="width:960px; margin:0 auto;">
    <table class="noborder3" style="margin:0;"><%=createdWith.getResource().getLink()%></table>
    </div>
<%
}
}
%>
<!----------------useWith---------------------------------->
<%
if(!resource.getUseWith().isEmpty()){
%>
    <p><b>This image might be used here:</b></p>
<%
    for (ResourceContainer useWith : resource.getUseWith()) {
    %>
    <div style="width:960px; margin:0 auto;">
    <table class="noborder3" style="margin:0;"><tr><td><%=useWith.getResource().getLink()%></td
    </div>
<%
}
}
%>
<!----------------seeAlso---------------------------------->
<%
if(!resource.getSeeAlso().isEmpty()){
%>
    <p><b>Related experiments, images or signals:</b></p>
<%
    for (ResourceContainer see : resource.getSeeAlso()) {
    %>
        <div style="width:960px; margin:0 auto;">
        <table class="noborder3" style="margin:0;"><tr><td><%=see.getResource().getLink() %></t
        </div>
<%
    }
}
```

```
335 private static void migrateContentImages(ResourceDetails resourceDetails, Resource resource) throws DBException {
336     try {
337         ContentPath contentPath = resourceDetails.getContentPathOfFirstImage();
338         resource.data.additional.Image newImage = new resource.data.additional.Image();
339         newImage.setPfad(contentPath.getHostURL() + contentPath.getBaseURL() + "/" + contentPath.getFolderPath() + "/");
340         newImage.setDateiname(contentPath.getFileName());
341         newImage.setEndung(contentPath.getFormat().getFormatString());
342         newImage.setResourceID(resource.getResourceID());
343         T_ImageDB4Servlet.createTImage(newImage);
344     } catch (Exception e) {
345         System.out.println(resourceDetails.getTitle() + " has no ContentPicture(s)");
346     }
347 }
```

Abbildung A.10: InitResource migrateContentImages()

```
65 public static String getT_wiringTitle(long myWiringId) throws ClassNotFoundException {
66     //Get connection to database
67     Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
68     Connection connection = DriverManager.getConnection("jdbc:ucanaccess://" +
69                                                         oldWiringsDatabaseAccessPath);
70         String selectString = "SELECT * FROM T_Wiring WHERE WiringId = ?";
71         Object[] parameterList = {myWiringId};
72         DBSelectCommand cmd = new DBSelectCommand(connection, selectString, parameterList) {
73             @Override
74             protected Object handleResultSet(ResultSet rs) throws Exception {
75                 while (rs.next()) {
76                    t_wiringTitle =  rs.getString("Title");
77                 }
78                 return null;
79             }
80     };
81     cmd.run();
82     connection.close();
83     return t_wiringTitle;
84 }
```

Abbildung A.11: LiveDatabaseManager getT_wiringTitle()

```
376 private static void migrateUseWith() {
377     for (Resource image : allDbImages) {
378         String dbTitle = image.getTitel();
379         for (ResourceDetails rd : allImages) {
380             String detailsTitle = rd.getTitle();
381             if (dbTitle.equals(detailsTitle)) {
382                 String useWithString = rd.getUseWithString();
383                 for (Resource wr : allDbWirings) {
384                     try {
385                         WiringResource wrr = (WiringResource) wr;
386                         String t_classname = wrr.getClassname();
387                         if (useWithString.equals(t_classname)) {
388                             T_LinkedResourcesDB4Servlet.createUseWith(image.getResourceID(),
389                                     wr.getResourceID());
390                         }
391                     } catch (NullPointerException e) {
392 
393                     } catch (DBException e) {
394                         throw new RuntimeException(e);
395                     }
```

Abbildung A.12: InitResources migateUseWith

Literaturverzeichnis

[1] Herbst, Robin
Bachelorarbeit: Online Repository fur Signale und Bilder - Suchfunktion, 2022

[2] `https://www.etti.unibw.de/labalive/`, 16.05.2023

[3] `https://www.namsu.de/Extra/befehle/Anfuehrungszeichen.html`, 16.05.2023

[4] `https://stackoverflow.com/questions/34218552/java-lang-classnotfoundexceptionnet-ucanaccess-jdbc-ucanaccessdriver`, 22.05.2023

[6] Albert Lattke: *Bachelorarbeit, 2022*

[7] Prof. Dr. Erwin Riederer: *Simulation kommunikationstechnischer Systeme, Einführung in labAlive*, kA, Stand: 14.04.2021

[8] `https://www.w3schools.com/howto/default.asp`, 25.06.2023

[9] `https://docs.oracle.com/javase/7/docs/api/overview-summary.html`, 03.07.2023

[10] `https://stackoverflow.com`, 12.05.2023

[11] `http://137.193.219.1:3000/leo/LabAlive-www`, 12.08.2023