TextGrid
Virtuelle Forschungsumgebung
für die Geisteswissenschaften

Blaupausen zur Integration zukunftsfähiger Entwicklungen (R 1.3.1 und M 1.3.2)

Version 2015
Arbeitspaket 1.3
verantwortlicher Partner Universität Würzburg

TextGrid

Virtuelle Forschungsumgebung für die Geisteswissenschaften

Ein Projekt im D-GRID

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

Projekt: TextGrid – Institutionalisierung einer Virtuellen Forschungsumgebung in den Geisteswissenschaften
BMBF Förderkennzeichen: 01UG1203B
Laufzeit: Juni 2012 bis Mai 2015

Dokumentstatus: <Final>
Verfügbarkeit: <öffentlich >
Autoren:
Thorsten Vitt (UWÜ)

Inhaltsverzeichnis:

1. Responsive Design für textgridrep.de

Die Benutzung von Smartphones und Tablets gerade zum Rezipieren von Inhalten hat in den letzten Jahren stark zugenommen. Für das Web-Frontend zum TextGridRep wurde deshalb das Design mit *Responsive-Design*-Techniken so weiterentwickelt, dass auf Smartphones und Tablets automatisch eine an die spezifischen Bildschirmformate angepasste Darstellung gewählt wird.

1.1. Hintergrund

Das TextGridRep-Webportal bietet das Durchsuchen und Blättern in den Inhalten des TextGridRep an. Für die Releases 1.0 und 2.0 wurde die Weboberfläche von einem professionellen Webdesigner für die Verwendung auf Desktop-Browsern gestaltet.

1.1.1. Technische Einschränkungen des Portals in Version 2.0

Aufgrund der Zielrichtung auf Desktops benutzt das Portal ein festes, 960 Pixel breites Layout für den Inhaltsbereich. Das ist sinnvoll, um die Zeilenlänge auf ein lesbares Maß zu beschränken, aber typischerweise zu breit für die Bildschirme von Handys und Tablets. Im Idealfall sind de Mobilgeräte in der Lage, in den Textbereich einzuzoomen und den Text zu reformatieren, im schlimmsten Fall jedoch passt der Inhalt einfach nicht auf den Bildschirm und der Benutzer muss vor- und zurückrollen.

1.2. Responsive Design

Der von Ethan Marcotte eingeführte[1] Begriff *Responsive Web Design* bezeichnet einen Ansatz im Webdesign, bei der die Seite sich an die äußeren Gegebenheiten wie etwa die Designbreite anpasst. Dieser Ansatz bietet einige Vorteile gegenüber separaten Mobilseiten: So wird das vorhandene Design nur dort verändert, wo dies aufgrund der Eigenheiten des Zielgeräts notwendig ist, die URLs bleiben gleich, und die Anpassung erfolgt tatsächlich anhand von relevanten Eigenschaften wie etwa der Bildschirmgröße und nicht anhand von Geräteklassen, damit ist der Ansatz auch in Bezug auf die Entwicklung neuerer Geräteklassen (mit z.B. anders dimensionierten Bildschirmen) flexibel.

1.2.1. Anpassung von textgridrep.de an Responsive Design – Minimalinvasiver Ansatz

Das Ziel der ersten Iteration der Anpassung von textgrid.de an Responsive Design war, alle Seiten auf aktuellen Smartphones und Tablets ohne vertikalem Scrollen nutzbar zu machen, ohne dabei das vorhandene Design zu sehr zu verändern.

Die Seite instruierte Zielgeräte bereits zu einer Zoomstufe von 100%, sodass noch die folgenden Schritte notwendig waren:

[1]Ethan Marcotte: Responsive Web Design. http://alistapart.com/article/responsive-web-design

Die feste Breite von 960px des Hauptbereichs wurde durch eine Obergrenze von 960px ersetzt. Dadurch werden die Zeilen auf großen Bildschirmen nicht zu lang, auf schmalen Bildschirmen wird jedoch bereits eher umbrochen.

Auf schmaleren Bildschirmen wurde das *padding*, der innere Rand der Hauptseite reduziert. Dazu werden CSS-Media-Queries[2] benutzt, mit denen bestimmte Regeln abhängig von Eigenschaften des Geräts wie der Bildschirmbreite aktiviert werden können.

Anpassung des Navigationsbalkens: Auf kleineren Geräten passt der Balken nicht mehr in eine Zeile, und das Registerkartendesign sah unnatürlich aus, wenn es in mehrere Zeilen umbrochen wurde. Hierfür wird deshalb auf schmalen Geräten ein eigenes Design verwendet.

Die Eingabefelder wurden ebenfalls an unterschiedliche Geräte angepasst. Da die Seite ein Plugin verwendet, das nicht mit Eingabefeldern variabler Länge zurechtkommt, werden eine Reihe von festen Breiten abhängig von der Bildschirmbreite verwendet.

Die Seitenleisten mit den Suchknöpfen auf den Such- und Ergebnisseiten rutschen auf schmaleren Bildschirmen nach unten und geben die gesamte Breite für den Hauptteil der Seite frei.

Eine TextGridRep-Seite auf einem typischen Smartphone-Display (nach den Anpassungen)

[2]CSS-Media-Queries: http://www.w3.org/TR/css3-mediaqueries/

2. E-Book-Export

Das TextGridRep dient gerade als Repositorium für größere Textmengen, zum Lesen dieser Texte ist jedoch der Bildschirm eines Desktop-Rechners nicht ideal. E-Book-Reader sind eine beliebte Alternative, um mit wenig Gewicht und einer auf das Lesen optimierten Displaytechnologie größere Textmengen zu rezipieren.

Das EPUB-Format[3] ist ein offenes Format für E-Books, das von den meisten E-Book-Readern und Tablets unterstützt wird. Für Amazons Kindle-Reader gibt es Konvertierungstools, die EPUBs in das proprietäre Kindle-Format wandeln.

Ein EPUB-E-Book ist im Wesentlichen eine Datei im ZIP-Format, die die eigentlichen Texte in einer XHTML-Version, Styling mit CSS und Bilder in den webüblichen Formaten wie JPEG enthält. Zusätzlich sind weitere Struktur- und Metadaten enthalten:

- Eine unkomprimierte Markerdatei mit dem Text application/epub+zip als erste Datei im Archiv,
- eine Container-Datei META-INF/container.xml,
- eine Strukturdatei (OPF) mit Dublin-Core-Metadaten und einem technischen Inhaltsverzeichnis *(spine)*, das die enthaltenen Dateien auflistet,
- und einem Inhaltsverzeichnis, mit dem der Benutzer durch den Inhalt navigieren kann.

2.1. EPUB-Generierung aus TEI

Die TEI-Stylesheets des TEI-Konsortiums[4] bieten eine Konvertierung von TEI nach EPUB an. Für das Angebot in TextGrid waren jedoch noch einige Nacharbeiten nötig.

Dazu zählt vor allem die Unterstützung für *Aggregationen*, also Zusammenfassungen Bündelungen mehrerer TEI-Dokumente. Da die Zerlegung z.B. der Daten der Digitalen Bibliothek auf der untersten Werkebene angesiedelt ist, ist hier jedes Gedicht ein Werk, und damit ein einzelnes TEI-Dokument. Für den Export eines Gedichtbands als ein E-Book ist es deshalb nötig, die einzelnen Gedichte wiederum zusammenzufassen.

Hierzu wurde zunächst eine erste Version des *Aggregator* entwickelt, ein Webservice, der die Aggregationsstruktur des TextGridRep ablaufen und aus einer Aggregation mit allen rekursiv enthaltenen Daten eine TEIcorpus-Datei erzeugen kann. Dieser Service wurde dann mit einem XSLT-Prozessor und den TEI-Stylesheets versehen, um die Konvertierung nach EPUB durchführen zu können.

Die Stylesheets erforderten noch einige Anpassungen, um einerseits mit der Corpusstruktur zurechtzukommen und andererseits für die in TextGrid enthaltenen Daten und Textsorten eine vernünftige Darstellung zu erzeugen. Dazu wurde ein Wrapperstylesheet geschrieben, das die TEI-Stylesheets konfiguriert und z.T. ihr Verhalten überschreibt.

Die Stylesheets erzeugen die Grundstruktur des EPUB-Formats, indem sie die HTML- und Metadaten-Dateien direkt aus den aus dem TextGridRep bezogenen TEI-Daten in eine tempo-

[3] EPUB-Format: http://idpf.org/epub
[4] TEI-Stylesheets des TEI-Konsortiums: https://github.com/TEIC/Stylesheets

räre Struktur auf dem Server des Webservice konvertieren. Die Links zu Bildern in den TEI-Dateien bleiben zunächst als textgrid:-Links enthalten.

Der EPUB-Modus des Aggregator-Service erzeugt also bei einer Anfrage nach einer EPUB-Version eines oder mehrerer TextGrid-Dokumente zunächst das Eingabeformat für die Transformation und ruft darauf die Stylesheets zur Transformation in die interne EPUB-Struktur auf. Aus diesem Dateibaum wird dann das spezielle EPUB-Archiv generiert, indem die XHTML- und Metadateien in das Archiv eingepackt und dabei Links zu Grafiken umgeschrieben werden. Die entsprechenden Grafiken werden dann ebenfalls eingepackt und dabei direkt aus dem TextGridRep gestreamt, sodass die Belastung durch temporäre Dateien minimiert wird.

Der Ansatz versucht, so früh wie möglich mit dem Ausliefern der konvertierten Inhaltsdaten zu beginnen um so die Wartezeit im Browser, aber auch die lokal zwischengespeicherte Datenmenge zu minimieren.

Die EPUB-Generierung ist seit Anfang 2013 in das TextGridRep integriert und kann mittlerweile nicht nur Einzelstrukturen aus beliebigen Ebenen der Daten ausliefern, sondern auch beliebige Zusammenstellungen aus Daten.

3. Webversion TextGridLab

Das TextGridLab bietet als *Rich Client* zwar ein leistungsfähiges und erweiterbares User Interface – und z.B. ein webbasierter XML-Editor mit dem Leistungsumfang von etwa oXygen existiert auch heute nicht –, Webtechniken haben allerdings in ihrer Leistungsfähigkeit deutlich zugenommen und bieten den Vorteil eines installationsfreien, plattformübergreifenden Zugangs. In diesem AP wurden deshalb Möglichkeiten evaluiert, eine Weboberfläche für den dynamischen Teil von TextGrid zu entwickeln. Dabei sollte insbesondere die Komplexitätsreduktion der Oberfläche eine Rolle spielen – die mit der Flexibilität und Anpassbarkeit der Oberfläche des TextGridLab einhergehende Komplexität erfordert einen gewissen Lernaufwand und wird insbesondere von gelegentlichen Nutzern bisweilen kritisiert.

3.1. Eclipse Remote Application Platform (RAP)

Eine zunächst naheliegende Implementierungsvariante ist die Eclipse Remote Application Platform (RAP)[5]. Dabei handelt es sich um ein Framework aus dem Eclipse-Projekt, dass auf Single-Sourcing einer Desktop- und einer Webapplikation beruht. Die Idee ist also, dass man eine Applikation nach den Grundsätzen der *Rich Client Platform (RCP)*, auf der auch das TextGridLab beruht, entwickelt, für die Webapplikation jedoch die entsprechenden RCP- durch RAP-Komponenten ersetzt und die Applikation auf einen Application Server deployed. Im Idealfall kann also weitestgehend derselbe Code für beide Varianten verwendet werden.

In der Praxis gibt es allerdings eine Reihe von Problemen, die gegen die Realisierung dieses Ansatzes sprachen:

[5]Eclipse Remote Application Platform (RAP): http://www.eclipse.org/rap/

Gerade die interessanten Komponenten des TextGridLab werden nicht unterstützt. Für die etwa den XML-Editor, den Text-Bild-Linkeditor und die Suchergebnisanzeige werden im TextGridLab Komponenten verwendet, die über die Fähigkeiten der Standardwidgets hinausgehen. Das Zeichnen der entsprechenden Komponenten wird teilweise durch Lab-Code selbst übernommen. Diese Komponenten werden dadurch nicht durch RAP unterstützt und müssten neu entwickelt werden.

Das Datenmodell des TextGridLab[6] beruht darauf, die TextGrid-Resourcen durch virtuelle Einträge im lokalen *Workspace* zu repräsentieren. Die TextGrid-Werkzeuge benutzen dann ebenso wie nicht auf TextGrid angepasste Eclipse-Tools die Eclipse-Resourcen-API[7], um mit diesen Resourcen zu interagieren. Das führt dazu, dass sowohl Tools, die nicht auf TextGrid angepasst sind, mit den TextGrid-Resourcen arbeiten können, als auch das mit den entsprechenden TextGrid-Tools auf lokalen Dateien gearbeitet werden kann. RAP unterstützt diese Eclipse-Resourcen-API[8] jedoch nicht, sodass sowohl die Interaktion mit dem TextGridRep als auch das Resourcenhandling der einzelnen Tools für die RAP-Lösung neu implementiert werden müsste.

Kritik an der GUI des TextGridLab richtet sich vor allem auf deren Komplexität, die der Flexibilität und Generizität geschuldet ist. Für eine Weboberfläche sollte deshalb eine stärker komplexitätsreduzierte Oberfläche geschaffen werden, die ggf. projektspezifisch angepasst wird und nicht aufgabenspezifische Funktionen nicht anbietet. Die Weboberfläche eng mit der Laboberfläche zu koppeln erscheint nicht als sinnvoller Ansatz.

Berücksichtigt man diese Aspekte, scheint eine Weboberfläche auf der Basis von RAP eher die Nachteile des eclipsebasierten Ansatzes (komplexe APIs, komplexe, wenig webtypische GUI) zu haben, ohne den Aufwand für die Entwicklung der Weboberfläche wesentlich zu reduzieren. Von der Weiterverfolgung dieses Ansatzes wurde deshalb abgesehen.

3.2. Implementierung als *Single-Page Webapp*

Intensiver verfolgt wurde die Implementierung der Weboberfläche[9] als *Single-Page Webapp*. Die Grundidee dieses Modells, das etwa auch von populären Webanwendungen wie Twitter verfolgt wird, ist, an den Browser zunächst eine Grundstruktur der Anwendung in Form einer statischen HTML-Seite mit Anwendungslogik in JavaScript auszuliefern. Im Browser der Anwender laufende JavaScript-Anwendungen bilden dann die Schnittstelle zwischen Webservices und Anwendern, indem sie relevante Daten von Webservices abrufen, daraus User-Interface-Elemente in HTML rendern, Interaktionen der Benutzer entgegennehmen und wiederum in Webservice-Aufrufe umsetzen.

Zu den Vorteilen dieses Ansatzes gehört, dass die Webanwendung im Browser rasch auf Interaktionen reagieren kann und nur diejenigen Daten nachladen muss, die als Reaktion auf die Interaktion nötig sind, es muss also nicht die ganze Seite neu generiert werden. Dazu kommt

[6]Datenmodell des TextGridLab: https://dev2.dariah.eu/wiki/display/TextGrid/TextGridLab+Data+Model
[7]die Eclipse-Resourcen-API:
http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcore%2Fresources%2Fpackage-summary.html
[8]RAP unterstützt die Eclipse-Resourcen-API nicht: http://stackoverflow.com/questions/18467725
[9]Implementierung der Weboberfläche: https://projects.gwdg.de/projects/labweb-exp/

eine klare Trennung von Funktionalität und Visualisierung. Für TextGrid wurde der Ansatz so implementiert, dass nach Möglichkeit auf die bereits für andere Services sowie das Lab verfügbaren Schnittstellen (TG-auth*, TG-crud, TG-search) zurückgegriffen wurde. Lediglich für komplexe Tätigkeiten, die sich nicht einfach und effizient im Browser lösen lassen (das Rendern der TEI-Dokumente nach HTML sowie der Export) wurde eine serverseitige Komponente implementiert – diese Komponente (der *Aggregator*) kann jedoch auch noch in anderen Kontexten genutzt werden, u.a. dient sie mittlerweile nach dem Umstieg des Repository von eXist auf ElasticSearch als Komponente zur Webdarstellung für das Repository.

3.3. Implementierte Grundstruktur und Herausforderungen

Die Grundstruktur der Webanwendung verwendet etablierte Bibliotheken und Frameworks aus dem Bereich der Webentwicklung, etwa:

- Bootstrap[10] als grundlegendes UI-Framework,
- das auch von Bootstrap benötigte jQuery[11] zur DOM-Manipulation,
- RequireJS[12] zur Abhängigkeitsverwaltung,
- Backbone.js[13] als MVC-Framework,
- Underscore[14] als Helfer zur funktionalen Programmierung.

Implementiert wurde zunächst ein Low-Level-Layer mit einem TG-Objekt zur Verwaltung von Konfigurations- und Anmeldeinformationen, das JavaScript-Kapselungen der grundlegenden APIs für TG-auth*, TG-crud und TG-search liefert.

3.3.1. TG-auth*

TG-auth* mit der Verwaltung der Benutzerrechte und -Rollen ist die einzige betrachtete Schnittstelle, die ausschließlich als SOAP-Interface vorliegt. Während SOAP im Webservicebereich verbreitet ist und z.B. in Java die automatische Generierung von Datenstrukturen und Clientcode aus der Servicebeschreibung bietet, ist dies im Bereich der Browseranwendungen eher unüblich. Die evaluierten SOAP-Bibliotheken für JavaScript erzeugen entweder (wie etwa wsdl2js aus CXF[15]) recht komplexen Code oder unterstützen die verwendete SOAP-Spielart nur begrenzt. Für das Frontend zu TG-auth*, das ohnehin eher einfache und gleichförmige Schnittstellen benutzt, wurde deshalb auf eine vollständige SOAP-Implementierung verzichtet und stattdessen eine sehr spezifische Kapselung implementiert, die die fertigen SOAP-Messages via Templating generiert und die Antworten als XML-Dokumente mittels jQuery parst.

[10]Bootstrap: http://getbootstrap.com/2.3.2/
[11]jQuery: https://jquery.com/
[12]RequireJS: http://requirejs.org/
[13]Backbone.js: http://backbonejs.org/
[14]Underscore: http://underscorejs.org/
[15]wsdl2js aus CXF: http://cxf.apache.org/docs/wsdl-to-javascript.html

Auf der Basis dieses Clients wurde dann ein Modell implementiert, das die Projekte, Personen und Rollen entsprechend modelliert und die Grundlage für die GUI bildet, und natürlich die entsprechende GUI in Form passender Views.

Backbone.js geht, wie bei dieser Art von Frameworks üblich, davon aus, dass die Synchronisierung zu einem JSON sprechenen Service wie etwa einer dokumentenorientierten Datenbank erfolgt. Es ist glücklicherweise möglich, die Synchronisierungsroutinen auf jeder Ebene zu überschreiben, sodass die Anbindung an die XML/SOAP-basierten Schnittstellen problemlos möglich war. Einfacher wäre es indes bei JSON-basierten REST-Schnittstellen gewesen.

3.3.2. TG-crud

TG-crud bietet mittlerweile neben der ursprünglichen SOAP- auch eine REST-Schnittstelle an, die ansonsten weitestgehend parallel zur SOAP-API funktioniert. Auch hier wurde eine Kapselung implementiert, die – ähnlich der Implementierung im TextGridLab – ein Modell zur Kapselung jedes TextGrid-Objektes bietet, mit entsprechenden Methoden für Zugriff und Manipulation.

Als Herausforderung erwies sich hier der Grundsatz, dass das Speichern von Daten und Metadaten gemeinsam erfolgen muss. Die REST-Schnittstelle für TG-crud verwendet hierzu Daten im MIME-Format multipart/related[16] mit einem Teil Meta- und einem Teil Inhaltsdaten. Eine im Browser lauffähige Bibliothek dafür musste erst entwickelt werden.

Ebenfalls der Aufmerksamkeit bedarf hier der Umgang der Browser mit Caching: GET-Requests per AJAX führen die Browser tatsächlich nur einmal pro URL aus, sofern nicht die die Anwendung hostende Seite neugeladen wird. Nach dem Speichern, bzw. um sicherzugehen, dass tatsächlich die aktuellste Version verwendet wird, muss deshalb bei Aufrufen an TG-crud oder Aggregator ein disambiguierender Parameter etwa mit einem Timestamp oder einer Seriennummer angehängt werden.

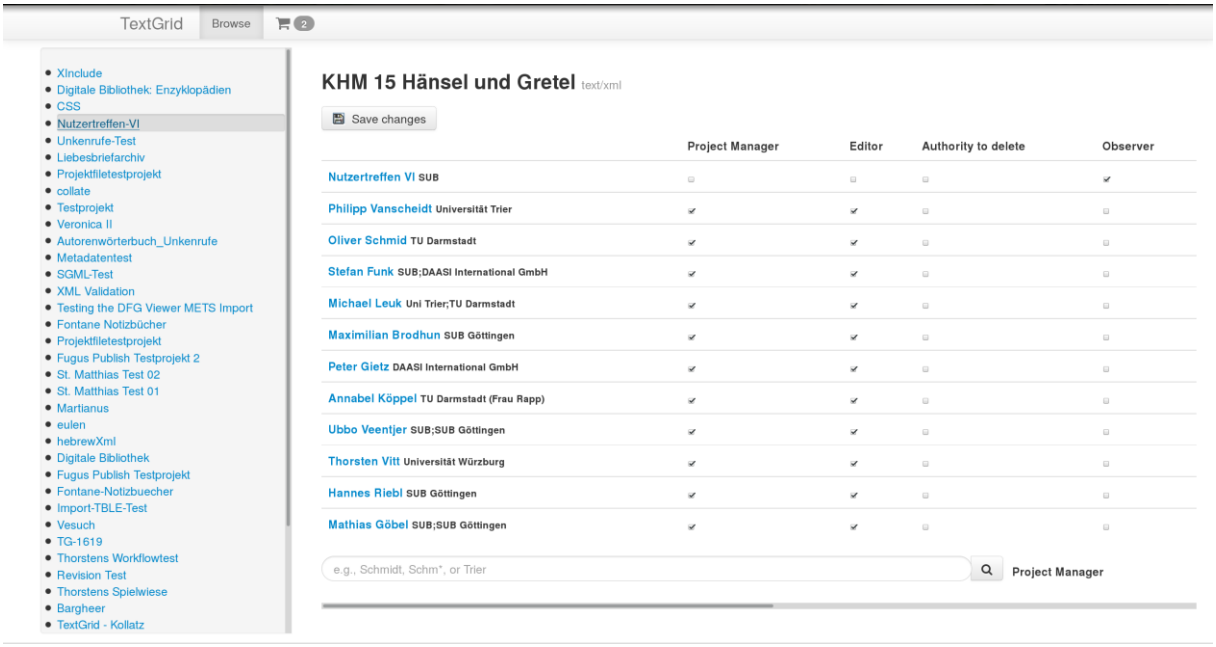
3.3.3. TG-search

TG-search ist von vornherein auch mit Hinblick auf Webanwendungen entwickelt worden – das TextGrid-Repository benutzt ebenfalls AJAX-Requests und stellt die Ergebnisse dar. Eine Integration war in dieser Hinsicht kein Problem. Implementiert wurden exemplarisch Wrapper für die bislang benötigten Funktionen, etwa zum Auflisten eines Projekts oder von Aggregationen. Eine zentrale Grundstruktur ist dabei ein Parser, der die XML-Suchergebnisse in eine Struktur aus TextGridObject-Wrappern wandelt, die die verschiedenen Zugriffsmöglichkeiten (auch via TG-crud) auf ein Objekt abstrahiert.

3.3.4. Grund-GUI

Eine exemplarische GUI wurde auf der Basis von Bootstrap[17] (als GUI-Framework) und Backbone[18] (als MVC-Framework) implementiert.

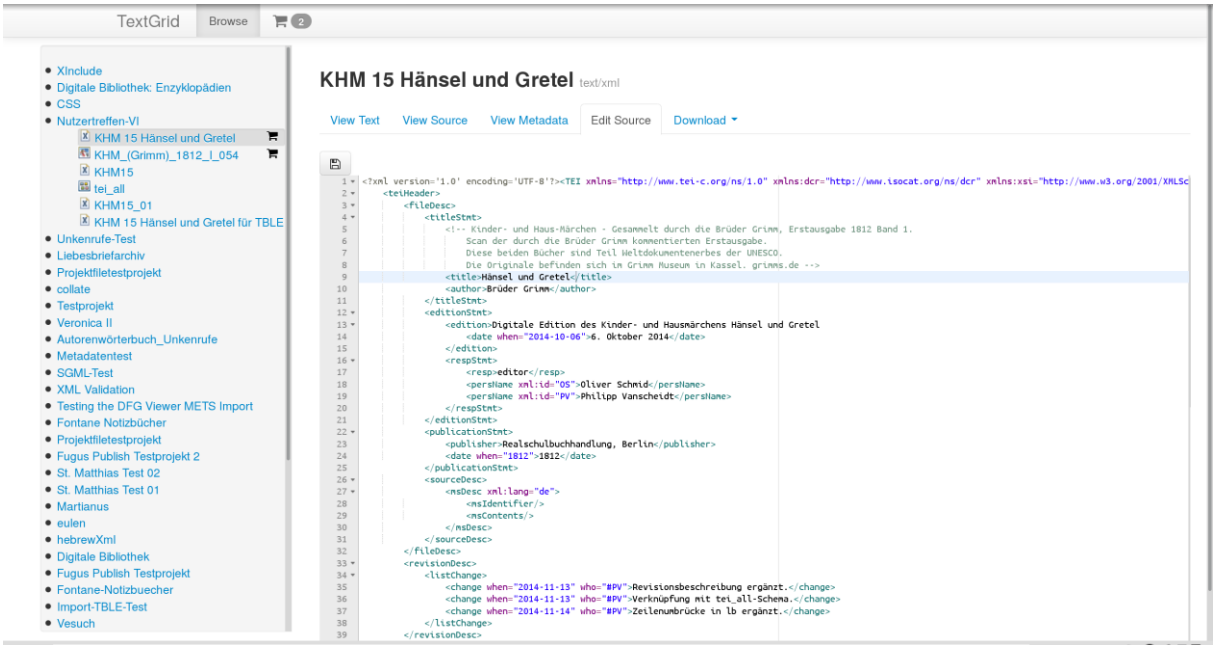
[16]multipart/related: https://tools.ietf.org/html/rfc2387
[17]Bootstrap: http://getbootstrap.com/2.3.2/
[18]Backbone: http://getbootstrap.com/2.3.2/



Exemplarische GUI: Rollenverwaltung

Die Basis ist ein *Navigator*, der – analog zum TextGridLab – die Projekte, auf die der Benutzer Zugriff hat, und die darin befindlichen Objekte in einer dynamischen Baumstruktur darstellt und dadurch navigieren lässt. Bestimmte Operationen wie die Aufnahme eines Objekts in den *Basket* zur gemeinsamen Weiterverarbeitung können direkt im Navigator durchgeführt werden.

Wird ein Objekt selektiert, so wird ein globales Event ausgelöst, auf das andere Teile der GUI reagieren können. Für den Hauptbereich der GUI wird dabei ein Plugin-basiertes User Interface angeboten, das abhängig vom Typ des selektierten Objekts unterschiedliche GUI-Zugänge in Form von dynamisch geladenen Tabs und Menüpunkten anbietet – zum Beispiel wird für Projekte eine Rechteverwaltung angezeigt, für XML-Dokumente stehen eine (gerenderte) HTML-Anzeige, eine Quelltextanzeige und ein XML-Editor zur Verfügung, für Aggregationen eine HTML-Anzeige, eine Eintragsliste sowie diverse Export-Optionen. Konkrete Plugins können dabei durchaus für unterschiedliche Objekttypen zur Verfügung stehen: So gibt es die Metadatenanzeige für alle Objekte, die HTML-Ansicht für XML-Objekte wie für Aggregationen, und den Texteditor für alle textbasierten Formate.



Exemplarische GUI: XML-Bearbeitung

3.4. Ausblick

Die Implementierung hat einen Ansatz gezeigt, wie eine Web-Oberfläche für den dynamischen Bereich von TextGrid funktionieren könnte, und sie bietet eine Reihe von Grundbausteinen an, die für produktive Versionen genutzt werden können. Der Ansatz erlaubt die Gestaltung einfacher, reaktiver, nicht mit Metafunktionalität überladener GUIs, und er erlaubt die Integration existierender webbasierter Tools.

Zu den offenen Herausforderungen gehört die Einbindung bzw. Implementierung konkreter, fachspezifischer Tools. So wurde etwa zum Bearbeiten von XML- und anderen Textdateien der recht leistungsfähige Source-Code-Editor ACE[19] eingebunden. Gerade in jüngster Zeit sind eine Reihe von Projekten entstanden, die XML-Editoren für das Web entwickeln (eine Übersicht[20] ist auf der Seite von Xonomy[21] zu finden), hier wäre zu prüfen, inwieweit ggf. projektspezifisch geeignetere Ansätze als ACE einzubinden sind. Auch für andere Tools wie etwa für den Text-Bild-Linkeditor hätte die Anpassung bzw. Entwicklung einer webbasierten Alternative den Rahmen dieser Studie gesprengt.

Ebenfalls zu den offenen Herausforderungen gehört eine praktikable und Shibboleth-kompatible Integration des Login-Mechanismus in die Webanwendung. Hintergrund ist, dass die Authentifizierung bei TextGrid-Diensten mithilfe einer *Session ID* erfolgt, die von einem TextGrid-SP nach der Authentifizierung mittels Shibboleth ausgegeben wird. Im TextGridLab funktioniert der Login so, dass in einem dedizierten Browser-Control der entsprechende Service Provider angesteuert, von dort aus zu den Shibboleth-Komponenten (Wayfinder, IdP der Institution des Benutzers) weitergeleitet und schließlich nach erfolgreichem Login die Session-ID aus der im Browser-Control angezeigten Ergebnisseite ausgelesen wird. Für die webba-

[19]ACE: http://ace.c9.io/
[20]Web-basierte XML-Editoren: http://www.lexiconista.com/xonomy/web-based-xml-editors/
[21]Xonomy: http://www.lexiconista.com/xonomy/web-based-xml-editors/

sierte Anwendung funktioniert dieses Verfahren jedoch nicht, da die Sicherheitsmechanismen der Browser das entsprechende Auslesen verhindern. Lösungen – etwa auf der Basis von OAuth[22] oder ECP[23] – erfordern Weiterentwicklungen in der TextGrid-AAI-Infrastruktur. Ggf. kann dies im Zuge von Weiterentwicklungen in DARIAH erfolgen – TextGrid- und DARIAH-AAI-Infrastruktur wurden Anfang 2015 zusammengeführt.

4. Zukunftsfähigkeit TextGridLab

TextGridLab 1.0 und 2.0 beruhen auf der Eclipse-Versionsreihe 3.x. Eclipse verfolgt bereits seit einiger Zeit eine neue, architektonisch umgebaute Versionsreihe 4.x, und mit dem aktuellen Eclipse 4.3 (Kepler) ist es erstmals zu einem *simultaneous release* ohne eine 3.x-Version gekommen.

Für das TextGridLab wurde die Migration auf die Eclipse-Version 4.3 negativ evaluiert, da in Verbindung mit dieser Eclipse-Version eine Reihe von Bugs auftraten, die nicht endgültig diagnostiziert und gelöst werden konnten. Ein technologisches Upgrade des bislang auf Eclipse 3.7 basierenden TextGridLab bis Projektende ist dennoch unausweichlich, auch um der technologischen Weiterentwicklung der Plattformen Rechnung zu tragen. Eine Portierung auf die zum Projektende aktuelle Version *Eclipse 4.4 (Luna)* konnte erfolgreich umgesetzt werden, so dass die für den Fall des Scheiterns vorbereitete Migration auf den letzten Stand der Eclipse-3-Reihe nicht zur Anwendung kommen musste.

5. Mehr Interaktivität im Lab mit ECF

Für das TextGridLab wurde testhalber das Eclipse Communication Framework (ECF)[24] integriert.

Nach der Installation des ECF ins TextGridLab können die Benutzer sich bei einem Chat-Anbieter (z. B. einem XMPP-Anbieter wie Jabber.org, Google Talk, GMX oder Rechenzentren) anmelden. So ist mit den entsprechenden Kontakten ein Chat direkt im TextGridLab-Fenster möglich.

Viel interessanter ist jedoch die DocShare-Funktion: Via Kontextmenü kann eine Benutzerin beliebige Editoren an ihren Kontakt senden. Der Editor erscheint dann auch im TextGridLab-Fenster ihres Kontakts, beide Parteien können gleichzeitig am selben Dokument arbeiten und live die Änderungen des Partners verfolgen. Neben Editoren können auch andere Elemente der Benutzeroberfläche im TextGridLab des Partners eingeblendet werden.

Da das ECF explizit auf Erweiterbarkeit durch andere Tools ausgelegt ist, sind mit ein wenig Programmierarbeit Szenarien denkbar, die die kollaborativen Fähigkeiten stärker ins Lab integrieren. So könnte z. B. mit einem selbst betriebenen XMPP-Server jedem TextGrid-Benutzer ein XMPP-Account zugeordnet werden und die Anmeldung am Server erfolgte dann automatisch mit der Anmeldung oder über einen „Online“-Knopf in der Toolbar.

[22] OAuth: http://oauth.net/
[23] ECP: https://wiki.shibboleth.net/confluence/display/CONCEPT/ECP
[24] Eclipse Communication Framework (ECF): http://www.eclipse.org/ecf

Entsprechend könnte beispielsweise für Projekte ein Gruppenchat angeboten werden, in dem alle zurzeit aktiven Projektmitglieder miteinander diskutieren. Ebenso könnte bei Bearbeitungskonflikten (A möchte ein Objekt bearbeiten, das B bereits geöffnet hat) das weitere Vorgehen über ein Chatfenster diskutiert und das Objekt z.B. im oben beschriebenen gemeinsamen Modus geöffnet werden; über die Anbindung z.B. eines TextGrid-weiten Benutzerchats wäre die Möglichkeit eines Live-Supports evtl. auch der Benutzer untereinander gegeben.

6. Massenexport aus dem Repository

Für die Entwicklung analytischer/Big-Data-Verfahren ist es üblich, Korpora zusammenzustellen, die dann exportiert und mit unterschiedlichen, oft von den Forschern selbst entwickelten lokalen Werkzeugen weiterverarbeitet werden. Um diese Anforderungen zu bedienen, wurde die Exportschnittstelle *Aggregator* weiterentwickelt.

Der Aggregator[25] ist in der Lage, die TextGrid-Aggregationsstruktur abzulaufen und die entsprechend aggregierten Objekte in verschiedene Formate zu konvertieren. Dazu gehört das bereits erwähnte EPUB sowie HTML in diversen Varianten etwa zur Verwendung in der Weboberfläche, aber auch explizit zur Weiterverarbeitung gedachte Formate: TEI-Corpus und ZIP.

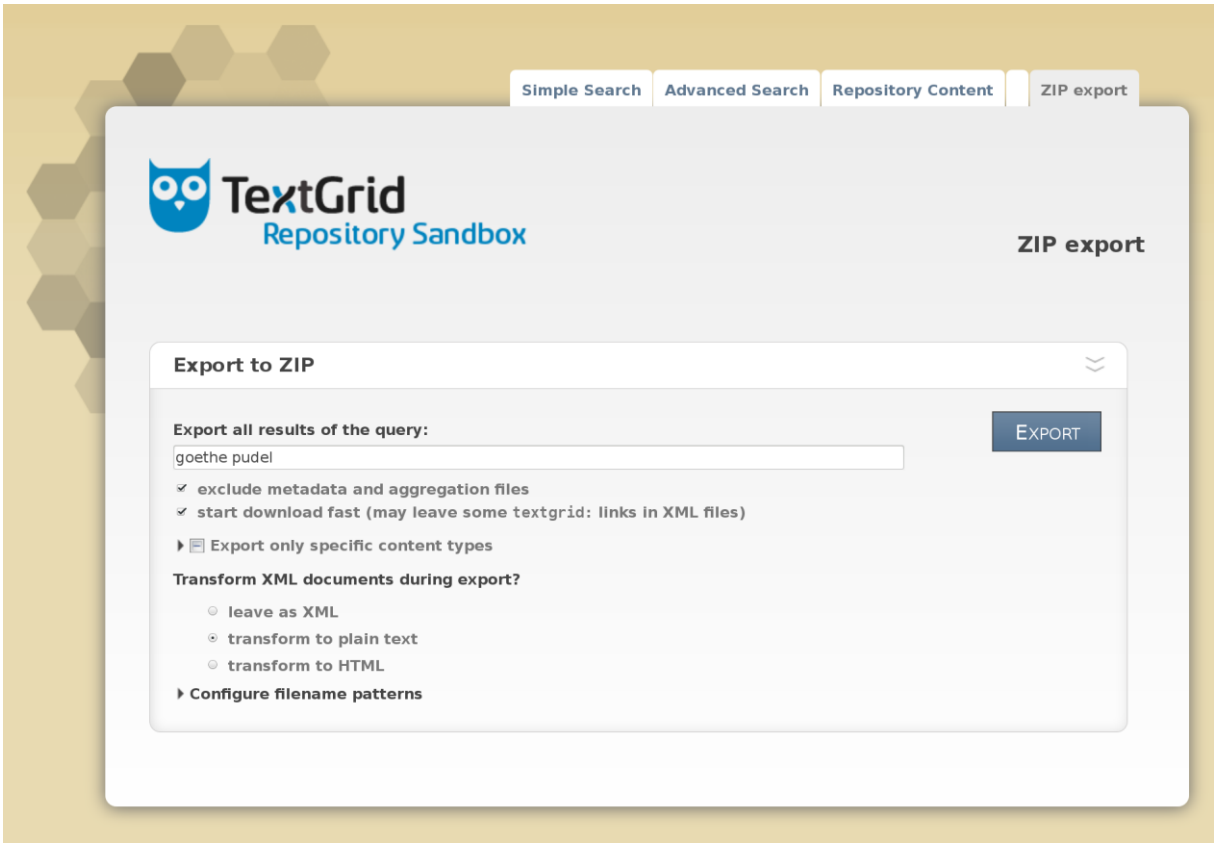
Der *TEI-Corpus-Export* sammelt lediglich die aggregierten TEI-Dokumente ein und generiert ein großes TEI-Corpus-Dokument daraus. Die Metadaten auf Aggregationsebene werden in TEI-Header konvertiert, Links und IDs so umgeschrieben, dass es nicht zu Duplikaten (und damit zu Verstößen gegen die Wohlgeformtheit) kommt. Der TEI-Corpus-Export ist recht schnell und kann seine Ergebnisse bereits streamen, während sie noch zusammengestellt werden. Er bildet auch die Grundlage für den HTML- und den EPUB-Export mehrerer Dokumente.

Der *ZIP-Export* exportiert in der Standardeinstellung alle rekursiv aggregierten Dokumente von der oder den Ausgangsaggregation(en) in der Form, in der sie im Repository vorliegen. Da TextGrid-Objekte keine Dateinamen haben, sondern lediglich opake URIs, besteht die Möglichkeit, Dateinamen und Verzeichnisstruktur in einem Wunschformat, das gewisse Metadatenfelder mit einbeziehen und dabei in ein »sicheres« ASCII-Subset konvertieren kann, zu vergeben. In Dokumenten bekannter Formate werden Links entsprechend umgeschrieben, sodass sie auf Dateinamen verweisen. Die Metadaten werden als XML-Dateien mitexportiert. In einer Datei, die vom Importtool des TextGridLab gelesen werden kann, wird die URI-Dateinamen-Zuordnung und die verwendeten Rewriting-Einstellungen festgehalten, sodass ein Reimport der Daten über das Lab einfach möglich ist.

Viele Tools, die gerade statistische Analysen auf Texten oder corpuslinguistische Aufbereitungen ausführen, arbeiten nicht mit TEI-Dateien, sondern lediglich mit reinem Text. Zur Unterstützung dieser Tools ist es möglich, direkt beim Export eine Konvertierung der TEI-Dateien aus dem TextGridRep durchzuführen. Dabei werden Tags und Header auf semantisch sinnvolle Weise entfernt. Der Mechanismus ist erweiterbar, im Prinzip kann man hier jedes beliebige XSLT-Stylesheet beim Export ausführen lassen und die Ergebnisse entsprechend zippen.

[25]Aggregator: http://dev.digital-humanities.de/ci/view/TextGrid%20Non-Lab/job/Aggregator/site/

Neben dem Export von Einzeldateien oder Aggregationen unterstützt der Aggregator auch den Export einer Liste von Objekten, wie sie etwa mit dem *Basket*-Feature des TextGridRep zusammengestellt werden kann, sowie sogar der Ergebnisse von Suchanfragen. Das letztgenannte Feature konnte etwa benutzt werden, um etwa ein Archiv aller Gedichte aus dem TextGridRep zu exportieren, die dann von Klemens Bobenhausens Metricalizer[26] metrisch analysiert wurden (und in der nächsten Version des Metricalizers zur Verfügung stehen werden). Eine experimentelle Weboberfläche für den ZIP-Export steht zur Verfügung.



Der Aggregator ist soweit modular aufgebaut, dass bei Bedarf weitere Exportformate ergänzt werden können, etwa METS[27] oder das in CLARIN von WebLicht verwendete TCF[28]. Der Service folgt grundsätzlich einem RESTful-Ansatz, es gibt keinen internen Zustand & Anfragen werden unmittelbar beantwortet. Die Verteilung im Sinne des High-Availability-Konzepts steht entsprechend nichts im Wege.

7. Anbindung der DARIAH-Pipeline

Im Rahmen des DARIAH-DE-Clusters 5 entsteht eine Pipeline[29] auf der Basis von DKpro, mit der Texte automatisch mit einer Reihe von linguistischen Annotationen versehen werden können, etwa zur Segmentierung, POS-Tagging, Lemmatisierung, Constituency- und Dependency-Parsing. Die Idee dabei ist, Forscher, die an der bloßen *Auswertung* annotierter Daten interessiert sind, von der Aufgabe, die zur Annotation notwendigen Einzeltools zusammenzu-

[26]Metricalizer: http://www.metricalizer.de/
[27]METS: http://www.loc.gov/standards/mets/
[28]TCF: http://weblicht.sfs.uni-tuebingen.de/weblichtwiki/index.php/The_TCF_Format
[29]Pipeline: https://dev2.dariah.eu/wiki/pages/viewpage.action?pageId=40213783

stellen, zu konfigurieren und zu verknüpfen, freizustellen und stattdessen eine nur minimal konfigurierbare Gesamtpipeline anzubieten, die aus Eingangsdaten im Text- bzw. TEI-Format (geplant) reichhaltig annotierte Daten in einem tabellarischen, tokenorientierten Format anbietet.

Die Pipeline liegt gegenwärtig in Form eines herunterladbaren, per Kommandozeile nutzbaren Javaprogramms vor. Es erscheint allerdings nützlich, gerade im TextGridRep veröffentlichte Texte, die für mehrere Forscher interessant sind, in einem Service aufzubereiten bzw. als aufbereitete Dateien zur Verfügung zu stellen.

Eine einfache Integration in den Aggregator (s.o.) scheidet allerdings aus: Aufgrund der umfangreichen und aufwändigen Analysen ist die Aufbereitung nicht nur speicher- sondern auch zeitintensiv. Es ist schon bei einem einzelnen, längeren Text mit einer Aufbereitungszeit zu rechnen, die so lange ist, dass ein Request, der erst nach der Transformation beantwortet ist, mit einem Timeout abbricht. Mit dem Restful-Ansatz des Aggregators ist dies deshalb nicht vereinbar.

Denkbar ist ein Caching-Ansatz:

Hierzu wird bei einem Request nach einer bestimmten Resource zunächst mit 202 Accepted[30] geantwortet und, etwa über ein Queuing-System[31], die Generierung der annotierten Version angestoßen. Ist die Annotation fertiggestellt, so wird ihr Ergebnis in einem Cache (z.B. Ehcache[32]) abgelegt und von künftigen Requests unmittelbar ausgeliefert, ohne eine erneute Annotation notwendig zu machen. Ggf. ist es denkbar, bei der Publikation von Dokumenten automatisch eine Aufbereitung anzustoßen und so mit bereits gefülltem Cache Anfragen sofort beantworten zu können.

Die Annotation ist nur auf der Ebene von Einzeltexten sinnvoll, nicht auf durch Aggregationen zusammengestellten Textsammlungen. Es kann jedoch sinnvoll sein, z.B. eine ZIP-Datei mit je einer Annotationsdatei pro Text in einer Aggregationsstruktur zurückzuliefern. Um das umzusetzen kann der Aggregator selbst als Client zum Annotationstool agieren.

[30] 202 Accepted: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.3
[31] Queuing-System: https://jqm.readthedocs.org/en/jqm-all-1.3.2/
[32] Ehcache: http://ehcache.org/