TBDis 3.0

Erweiterung des Online-Debattiertools um einen führenden Modus

– Bachelorarbeit im Fach Wirtschaftsinformatik –

Institut für Informatik
Universität Zürich

Betreuender Dozent
Prof. Dr. Lorenz Hilty

Eingereicht von
Daniel Lakic
Zürich, ZH, Schweiz
Matrikelnummer 10-728-509

Zürich, 16.08.2015

Inhaltsverzeichnis

1 Einleitung

Entscheidungen zu treffen, ist ein grosser Bestandteil unseres Lebens. Ständig müssen sie gefällt werden: Sei es etwas Belangloses, wie wo man in der Mittagspause essen geht, oder aber etwas Wichtigeres, wie welchen Karriereweg man einschlägt. Bei grösseren Entscheidungen, die man alleine trifft, erstellt man vielleicht eine Pro- und Kontraliste, um die verschiedenen Möglichkeiten besser abzugrenzen sowie die positiven und negativen Aspekte strukturieren zu können. Genauso wie man Entscheidungen alleine trifft, gibt es Fälle, in denen man sich gemeinsam in einer Gruppe zwischen den Alternativen festlegen muss. Dies kann sowohl vorteilhaft sein als auch zusätzliche Hürden mit sich bringen. Einerseits erhält man in einer Gruppe die verschiedensten Blickwinkel bezüglich eines Problems und bekommt somit eine Sichtweise, die man als Einzelperson nicht betrachtet hätte. Andererseits können diese diversifizierten Meinungen einen Konsens – und somit die finale Entscheidung – erschweren. Zusätzlich müssen Sitzungen abgehalten werden, um die Denkweisen und Meinungen der verschiedenen Mitglieder der Gruppe eruieren zu können. Bereits das kann sich jedoch oftmals als schweres Unterfangen erweisen: Sei es wegen der schieren Unmöglichkeit, einen passenden Termin für alle Gruppenmitglieder zu finden, oder aber der Schwierigkeit, die verschiedenen Gedankengänge sinnvoll und strukturiert in die Debatte einzubringen und zu protokollieren. TBDis („to be discussed“) bietet eine Lösung, um all diesen und noch weiteren Problemen, die bei einer solchen Debatte entstehen können, Abhilfe zu schaffen.

1.1 Zielsetzung dieser Arbeit

TBDis bietet eine stabile Plattform, um Debatten in Gruppen unterschiedlichster Grösse und zu den verschiedensten Themen zu führen. Die Logik, die hinter der Plattform steht, ermöglicht es, den Prozess der Entscheidungsfindung möglichst effizient zu gestalten. Um dies zu bewerkstelligen, verlangt die Applikation vom Nutzer jedoch, dass er diese Logik versteht und auch während der Debatte penibel anwendet. Verschiedene Versuche haben ergeben, dass den Nutzern diese Logik nicht trivial erscheint und folglich die korrekte Nutzung ausblieb. Deshalb ist das Kernziel dieser Arbeit, TBDis und seine Handhabung dem Nutzer näher zu bringen, damit die Applikation möglichst ohne grossen Aufwand angewendet werden kann. Momentan unterstützt TBDis zwei verschiedene Darstellungsmodi. Die Debatte kann in einer Baum- oder Listenstruktur dargestellt werden. Diese Arbeit erweitert die Applikation um ei-

nen weiteren Modus – den führenden Modus. Dieser Modus hat die Aufgabe, den Nutzer durch eine Debatte zu führen, sodass er sich nicht um die Logik kümmern muss. Dabei ist es wichtig, dass dieser Modus nur als Richtlinie gilt. Der Nutzer hat immer noch die Möglichkeit, die Aufforderungen zu ignorieren und die Applikation stattdessen nach Belieben zu nutzen. Mithilfe eines Dialogfensters wird während einer Debatte die nächste erwartete Aktion erläutert und die verschiedenen Handlungsoptionen aufgezeigt. Die Benutzerfreundlichkeit spielt dabei eine grosse Rolle. Untersuchungen anderer Group Decision Support Systems (GDSS), die in dieser Arbeit aufgezeigt werden, führen zu diesem Schluss. Ziel ist es, durch Anwendungen von bekannten Prinzipien des Designs von Web-Applikationen eine möglichst benutzerfreundliche Plattform zu schaffen. Mit anderen Worten: Die Debatte, und nicht die Applikation, soll im Zentrum stehen.

1.2 TBDis

Wie bereits in der Zielsetzung erwähnt, ist TBDis ein Online-Debattiertool. Es ist konzipiert, um Debatten jeglicher Grössen möglichst effizient und übersichtlich zu führen. Debatten in TBDis basieren auf der Aussagenlogik. Diese beschäftigt sich mit der Verknüpfung von Aussagen und bewertet sie nach deren Wahrheitsgehalt (Rautenberg, 1979, S. 1). Den Start einer TBDis-Debatte bietet die sogenannte Haupthypothese, auch Hauptfrage oder Thema genannt. Die Haupthypothese beschreibt das Kernproblem der Debatte, welches in der Gruppe gelöst werden muss, und ist gleichzeitig eine Aussage. Aufbauend darauf liefern die Nutzer Argumente, die für oder gegen die Aussage sprechen, welche durch weitere Argumente bewertet werden können. Dies führt dazu, dass die Debatte als stets wachsender Baum dargestellt wird. Argumente bestehen jeweils aus einer Reihe von Aussagen, die möglichst alle wahr sein müssen, um ein überzeugendes Argument zu liefern („TBDis Help-Wiki“, 2012). Aus Sicht der Aussagenlogik bildet ein Argument mit seinen Aussagen eine Konjunktion, was bedeutet, dass das Konstrukt von Aussagen nur wahr ist, falls alle Aussagen wahr sind. Abbildung 1 zeigt ein Beispiel einer solchen Debatte in TBDis und erläutert die verschiedenen Elemente. Der Standardmodus stellt die Debatte in der klassischen Baumstruktur dar, wobei die Hauptfrage den Hauptknoten des Baumes darstellt und die Argumente dazu die inneren Knoten und Blätter bilden. Eine kaskadierende Listenstruktur bildet die zweite Darstellungsmöglichkeit. TBDis stellt dabei die inneren Knoten und Blätter eingerückt dar, die sich mithilfe des Plus- und Minussymbols ein- und ausblenden lassen.

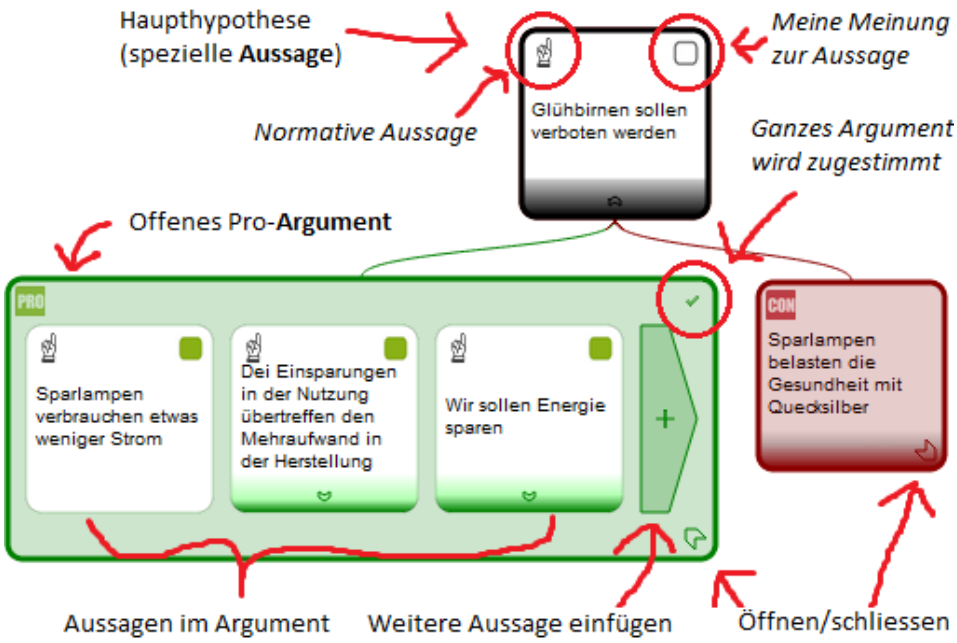

Abbildung 1: Beispiel einer TBDis Debatte inklusive Beschriftung der einzelnen Elemente in Baumstrukturform. Grafik von tbdis.org übernommen.

1.3 Entstehungsgeschichte und bisherige Entwicklungen

TBDis ist ein Prototyp, der auf der Theorie des „CANDis – Computer-Aided-Normative-Discourse“ Projektes aufbaut. Das Projekt wurde 2010 am Lehrstuhl für Informatik und Nachhaltigkeit der Universität Zürich in Zusammenarbeit mit der Abteilung Technologie und Gesellschaft der Empa durchgeführt (Hilty, 2010, S. 1). Aufbauend darauf wurde im Rahmen einer Bachelorarbeit ein erster Prototyp entwickelt (Kümin, 2010) und 2012 um weitere Funktionen erweitert, wie bspw. einem Reportgenerator, der Statistiken zu den Debatten liefert (Kümin, 2012). Dieser Prototyp bot eine fundierte Bandbreite an Funktionen, erwies jedoch noch einige Schwächen im Bereich der Sicherheit. Um die Applikation gegen Angriffe wie „Cross-Site-Request-Forgery“ (CSRF), „SQL-Injection“ oder „Cross-Site-Scripting“ (XSS) zu schützen, wurde das Projekt auf das PHP Symfony Framework portiert (Kübler, 2014). Bei der Portierung konnten die meisten bisher vorhandenen Funktionen übernommen werden. Lediglich die To-do-Liste, die dem Nutzer einen Überblick über die anstehenden Aufgaben verschaffte, wurde nicht implementiert. Der Algorithmus, der für die Generierung der To-do-Liste zuständig ist, bietet die Basis für den führenden Modus, der im Rahmen dieser Arbeit entwickelt wurde.

2 Group Decision Support Systems als Basis für TBDis

In einer Gruppe kommt es häufig zu Schwierigkeiten, wenn es darum geht, während des Prozesses der Entscheidungsfindung zu einem Konsens zu gelangen. Bereits das simple Problem der Terminfindung für die Gruppendiskussion oder ein komplexeres Problem, wie bspw. eine Informationsasymmetrie zwischen den Teilnehmern, erschweren es, eine geeignete Lösung zu finden. Group Decision Support Systems (GDSS) haben die Aufgabe, eine Gruppe bei einem solchen Prozess zu unterstützen. GDSS sind eine Variation der Decision Support Systems (DSS). Der Zweck von DSS ist es, einem Entscheidungsträger bei einer Entscheidungsfindung zur Seite zu stehen. Es filtert relevante Informationen und bietet die Möglichkeit, die Auskunft auf eine Art und Weise aufzubereiten, die den Nutzer bei der Lösung der Problemstellung assistiert (Burstein, 2008, S. 163). Das DSS dient dabei lediglich als Informationsquelle. Die eigentliche Entscheidung zu treffen, liegt folglich immer noch im Ermessen des Nutzers. GDSS gehen einen Schritt weiter und bieten die zusätzliche Funktion, solche Entscheidungen in Gruppen zu vereinfachen, indem sie die Kommunikation unterstützen. So ermöglichen sie bspw. eine asynchrone oder gar simultane Kommunikation, da das System von verschiedensten Orten, zu verschiedensten Zeitpunkten genutzt werden kann. Während DSS auf bereits bekanntes Wissen bzw. Daten und mathematische Berechnungen setzt, bauen GDSS auf den Input der Mitglieder auf und haben die zusätzliche Funktion, Menschen zu vernetzen (Burstein, 2008, S. 371-375). Tabelle 1 zeigt die verschiedenen Arten von GDSS und gibt einige Beispiele dazu wieder.

	Gleiche Zeit (synchron)	**Verschiedene Zeiten (asynchron)**
Gleicher Ort	***Gebrauch:*** *Face-to-Face Meetings* **Beispiele:** • Projektoren • Gruppenunterstützungssysteme	***Gebrauch****: Administratives, Archivierung und Filterung* **Beispiele:** • Shared Files • Gruppen Displays
Verschiedene Orte	***Gebrauch:*** *Cross-Distance Meetings* **Beispiele:** • Konferenzgespräche • Screen-Sharing	***Gebrauch:*** *Fortlaufende Koordination* **Beispiele:** • Gruppen Voicemail • Formular Management

Tabelle 1: GDSS Typen, angelehnt an: (Johansen, 1991)

2.1 Funktionen von GDSS

Burstein (2008) definiert folgende Kernfunktionen der GDSS:

- Ideengenerierung, -organisation und -priorisierung
- Anonymität
- Identifikation der Stakeholder
- Kommunikation zwischen den Beteiligten
- Organisational Memory
- Online Umfragen
- Management der Sitzung
- Bedienbarkeit

2.1.1 Ideengenerierung , -organisation und -priorisierung

Die verschiedenen Inputs eines jeden Mitglieds werden in einer GDSS Software gelistet und erlauben so wiederum eine bessere Übersicht über die Problemstellung. Der gewonnene Überblick hilft dem Ideengenerierungsprozess ungemein. Zusätzlich bietet ein solches System

eine Funktion, um die verschiedenen Inputs in einem schlüssigen Zusammenhang zu stellen und besonders wichtige Punkte hervorzuheben. Beiträge, die aufeinander aufbauen, werden zusammengefasst, inhaltlich identische bzw. irrelevante gelöscht und solche, die bei einer Entscheidungsfindung einen hohen Stellenwert besitzen, können hervorgehoben werden.

2.1.2 Anonymität

GDSS bieten oftmals die Möglichkeit, Ideen und Inputs anonym anzugeben, was den Fokus auf den Beitrag selber lenkt und nicht auf die Person, die ihn tätigt. Diskriminierungen können hiermit ausgeschlossen werden.

2.1.3 Identifikation der Stakeholder

Während des Prozesses der Entscheidungsfindung sind oftmals mehrere Stakeholder involviert, die identifiziert werden müssen. Module zu Stakeholderanalysen basierend auf Mason und Mitroffs Theorie (1981) werden üblicherweise in GDSS inkorporiert.

2.1.4 Kommunikation zwischen den Beteiligten

Ähnlich wie bei einem Chat ermöglichen GDSS einfache Kommunikation zwischen den Beteiligten, sei sie öffentlich oder privat.

2.1.5 Organisational Memory

Vergangene Meetings werden protokolliert und im System abgespeichert, um jederzeit die Informationen abrufen zu können. Zusätzlich lassen sich strukturierte Reporte dazu generieren, die statistische Informationen zum Input aller Mitglieder liefern.

2.1.6 Online Umfragen

Kurze Umfragen, um die Meinung zu einem bestimmten Thema zu erhalten, können erstellt werden.

2.1.7 Management der Sitzung

Alles rund um die eigentliche Sitzung wird hier betreut, sei es der Zeitpunkt für das nächste Meeting oder aber die Beschaffung von Hintergrundinformationen für die Problemstellung. Zusätzlich müssen Erkenntnisse aus jeder Sitzung zusammengefasst und die Sitzung initiiert werden.

2.1.8 Bedienbarkeit

Eine GDSS Software sollte möglichst ohne grosse Einarbeitungszeit von allen Nutzern bedienbar sein. Schwer bedienbare Software hindert den Prozess der Entscheidungsfindung, da mehr Zeit in die korrekte Bedienung als in die Entscheidungsfindung investiert wird. Somit hat benutzerunfreundliche Software trotz grossem Funktionsumfang den gegenteiligen Effekt auf dessen eigentlichen Zweck.

2.2 Vorhandene Funktionen in TBDis

Im Kern baut TBDis auf der Theorie von Group Decision Support Systems auf. Folglich sind viele der oben genannten Funktionen in dieser Applikation vorhanden. Die Ideengenerierung, -organisation und -priorisierung wird in TBDis jeweils durch die Darstellung der Haupthypothese, den Argumenten sowie deren Leitaussagen und Bedingungen erfüllt. Die logische Verknüpfung dieser Aussagen und die Positionierung der Argumente erlauben eine Organisation der verschiedenen Beiträge, die zugleich in einer Baumstruktur abgebildet werden. Zusätzlich bietet TBDis die Möglichkeit, über die Relevanz der Aussagen abzustimmen, womit die Priorisierung ebenfalls abgedeckt wäre. Anonymität ist ebenfalls gewährleistet: TBDis bietet die Möglichkeit eine Debatte mit Pseudonymen statt richtigen Namen zu führen. Geführte Debatten werden in TBDis abgespeichert und lassen sich mithilfe eines Befehls zu einem Report zusammenfassen, womit auch die Organisational Memory und das Management der Sitzung implementiert wären.

2.3 Nicht oder unzureichend umgesetzte Funktionen in TBDis

Einige der in 2.1 aufgelisteten Funktionen wurden in der bisherigen Entwicklung von TBDis ausser Acht gelassen. Entweder waren sie für die Kernfunktionalität der Applikation nicht

allzu relevant oder ihr Fehlen wurde erst zu einem späteren Zeitpunkt entdeckt. Eine Stakeholderanalyse (dazu Ziffer 2.1.3) kann bei TBDis nicht durchgeführt werden. Da TBDis nur für Debatten genutzt wird und somit die Aussagen im Vordergrund stehen, ist die Analyse der beteiligten Parteien nicht Zweck von TBDis. Es ist auch nicht möglich, online Umfragen zu starten oder privat mit anderen Beteiligten der Debatte zu kommunizieren. Die Kommunikation findet stets unter den Beteiligten der Debatte statt. Anderweitige Umfragen sind nicht notwendig, da in TBDis ohnehin die Meinung jedes Nutzers zu jeder Aussage abgefragt wird. Das Kernproblem von TBDis ist jedoch die Benutzerfreundlichkeit. Verschiedene Tests haben ergeben, dass vielen Nutzern unklar war, wie TBDis eigentlich zu nutzen ist (Hilty, persönliche Kommunikation, Februar 2015). Im Rahmen der Bachelorarbeit von José Kümin wurde ein Moderationsalgorithmus erstellt, der für jeden Nutzer eine To-do-Liste generiert (Kümin, 2010). Ziel dieser Liste war es, den Nutzer über den erwarteten Input zu informieren, womit der Punkt der Bedienbarkeit zumindest zu einem Teil realisiert wäre. Diese Implementation der To-do-Liste wurde während der Bachelorarbeit von Niels Kübler nicht übernommen. Sein Ziel war, die Applikation auf das Symfony-Framework zu portieren, um somit die Sicherheit zu verbessern (Kübler, 2014).

2.4 Fallstudien und Experimente zur Nutzung von GDSS

Einige Fallstudien zu GDSS wurden bereits durchgeführt. So analysierten Jerry Fjermestad und Starr Roxanne Hiltz (2000) 54 verschiedene Fallstudien, die über zwei Jahrzehnte publiziert wurden. In ihrer Analyse wurde zwischen DSS (reinen Decision Support Systems), GSS (bzw. GDSS) und CMC (Computer-Mediated Communication) unterschieden. CMC beschreibt jegliche Art von Kommunikation zwischen Menschen, die mithilfe von Computern geführt werden (Thurlow, Lengel, & Tomic, 2011, S. 14). 81.5% der untersuchten Fallstudien nutzten GDSS. Die Gruppen setzten sich aus mindestens 3 Mitgliedern zusammen, wovon 92% aus einem professionellen Arbeitsumfeld kommen.

Table 3. Factors Model Outcome Factors (Unit of Measure Is Measure)

1. Efficiency	
Measures	
Improved	28
Did not improve	1
No difference	1
No Measures	24
2. Effectiveness Measures	
Improved	39
Did not improve	2
No difference	3
No Measures	10
3. Satisfaction Measures	
Process Satisfaction	
Improved	20
No difference	2
No Measures	32
Outcome Satisfaction	
Improved	11
Did not improve	2
No difference	1
No Measures	33
Participation	
Improved	16
Did not improve	3
No difference	0
No Measures	35
4. Consensus Measures	
Improved	8
Did not improve	3
No difference	0
No Measures	43
5. Usability Measures	
Improved	12
Did not improve	2
No difference	1
No Measures	39

Improved:

Process support	Flexibility
Process structure	Divergent and convergent
Task structure	Enriched communication
Information exchange	Improved focus
Role perceptions	Increased number of ideas
Communication	Reduced stress
Number of ideas	Knowledge and knowledge sharing
The ability to deal with task complexity	High trust groups are effective and have active participation
Cohesiveness	

Abbildung 2: Ergebnisse der Untersuchung (Fjermestad & Hiltz, 2000)

In 28 von 54 untersuchten Fällen wurde eine eindeutige Verbesserung der Effizienz festgestellt im Vergleich zu konventionellen Meetings. In diesen erfolgreichen Fällen wurde der Prozess der Entscheidungsfindung um einige Wochen beschleunigt. Die Wirksamkeit der Meetings wurde in 39 von 54 gemessenen Fällen mithilfe von GDSS verbessert. So wurde bspw. ausgesagt, dass die Nutzung solcher Software zu tatsächlichen finanziellen Einsparungen führte. Im Allgemeinen trug die Nutzung einer solchen Softwarelösung zu verbesserter Kommunikation und reicherem Ideenaustausch. Sitzungen wurden effektiver geführt, da der Prozess der Entscheidungsfindung effizienter gestaltet wurde. Abbildung 2 zeigt die kompletten Ergebnisse der Untersuchung und was dabei tatsächlich verbessert wurde. Um diese positiven Ergebnisse auch auf TBDis übertragen zu können, muss die Benutzerfreundlichkeit verbessert werden. Gerade die Effizienz leidet, wenn viel Zeit aufgewendet werden muss, um die Logik hinter der Applikation zu verstehen. Um dieses Problem zu beseitigen, wird der führende Modus eingeführt.

3 Aufbau des führenden Modus

Der führende Modus hat die Funktion, dem Nutzer Vorschläge während einer TBDis-Debatte zu geben. Führt ein Nutzer eine Aktion aus, wie zum Beispiel das Erstellen eines neuen Pro-Arguments, erhalten die anderen Teilnehmer der Debatte einen Aufruf, dieses Argument nach dessen Wahrheitsgehalt zu bewerten. Ruft ein Teilnehmer die Debatte auf, der noch keine Meinung zum Argument abgegeben hat, erhält er sogleich ein Dialogfenster angezeigt. In diesem Fenster erhält er die Information, dass er das Argument noch nicht bewertet hat und erhält die Möglichkeit, die Bewertung sogleich auszuführen. Der führende Modus arbeitet mit Kümins Implementation des Moderationsalgorithmus zusammen, der für die Erstellung von To-do-Listen-Einträgen zuständig ist. In dieser Implementation werden sieben verschiedene Fälle (dazu Ziffer 3.1) eingeführt, die je eine andere Aktion seitens des Nutzers erfordern. Um dem Nutzer im führenden Modus eine Aktion vorzuschlagen, wird von der To-do-Liste das oberste Element – das Element mit der grössten Priorität – genommen. Je nach To-do-Listen-Fall werden entsprechende Dialogfenster angezeigt, in denen der Benutzer die vorgeschlagene Aktion durchführen kann. Je nach gewählter Handlungsoption kann es vorkommen, dass man zu einem anderen Fenster geführt wird, das dem Nutzer weitere Handlungsoptionen anbietet. Nach der Zustimmung einer Aussage wird dem Nutzer bspw. die Erstellung eines weiteren Pro-Arguments angeboten. Mit den verschiedenen Fällen wird dadurch ein Transaktionssystem umgesetzt. Eine Transaktion ist abgeschlossen, sobald alle Fenster für einen bestimmten Fall vom Nutzer bearbeitet wurden. Der nächste Fall wird im Anschluss der abgeschlossenen Transaktion geladen. Es besteht jederzeit die Möglichkeit, den Vorschlag abzulehnen und in einen der bereits bestehenden Modi seine gewünschten Aktionen durchzuführen. Falls der Nutzer wieder einen Tipp zur nächsten Aktion benötigt, kann er jederzeit in den führenden Modus wechseln, in dem dann das nächste Fenster für ihn angezeigt wird. Aktionen, die ausserhalb des dieses Modus durchgeführt werden, werden weiterhin in der To-do-Liste aufgefasst.

3.1 Moderationsalgorithmus und seine Fälle

Der Moderationsalgorithmus hat zur Hauptaufgabe, To-do-Listen-Einträge für die jeweiligen Nutzer zu generieren. Dabei bieten die Funktionen *todoList_tell()* und *todoList_untell()* das Kernstück des Algorithmus. Die Aufgabe der Funktion *todoList_tell()* ist es, Einträge in die

Datenbank zu schreiben. Mit der Funktion *todoList_untell()* werden diese wieder aus der Datenbank entfernt. Als Parameter werden die ID des Nutzers, die ID des Knotens und der jeweilige Fall übergeben. Dieser Fall beschreibt, welche Aktion vom Nutzer als Nächstes erwartet wird. In der nachfolgenden Tabelle werden die unterschiedlichen Fälle erläutert.

Nr.	**Fall**	**Auslöser**
1	judge	Der Nutzer hat noch keine Meinung zur Haupthypothese abgegeben.
2	judgeArgument	Der Nutzer hat noch keine Meinung zur Leitaussage eines neuen Arguments abgegeben.
3	judgePremise	Der Nutzer hat noch keine Meinung zu einer neuen Zusatzbedingung, die zu einem bestehenden Argument hinzugefügt wurde, abgegeben
4	reconsiderDue2Edit	Der Autor der Aussage hat die Formulierung bearbeitet. Nun kriegen andere Nutzer die Möglichkeit, die Aussage erneut zu bewerten.
5	reconsiderDue2AgreePro	Widerspruch, der dadurch entsteht, dass man einem Pro-Argument zu einer Aussage mit all seinen Zusatzbedingungen zugestimmt hat, jedoch die Aussage selbst abgelehnt hat.
6	reconsiderDue2AgreeContra	Widerspruch, der dadurch entsteht, dass man einem Contra-Argument zu einer Aussage mit all seinen Zusatzbedingungen zugestimmt hat, jedoch auch der Aussage selbst zugestimmt hat.
7	reconsiderDue2Inconsistency	Widerspruch, der dadurch entsteht, dass man sowohl einem Pro-Argument als auch einem Contra-Argument zu einer Aussage mit all seinen Zusatzbedingungen zugestimmt hat.

Tabelle 2: Fälle von To-do-Listen-Einträgen, angelehnt an: (Hilty, 2011)

Die Fälle *judge, judgeArgument, judgePremise* und *reconsiderDue2Edit* wurden bei Kümins Version durch einfache *todoList_tell()*- und *todoList_untell()*-Aufrufe den jeweiligen Nutzern der Debatte zugeordnet, sobald eine neue Aussage erstellt oder eine bestehende bearbeitet wurde. Für die Fälle *reconsiderDue2AgreePro*, *reconsiderDue2AgreeContra* und *reconsiderDue2Inconsistency* wurden zusätzliche Algorithmen entwickelt. Dabei wird die Funktion *wrapUp()* ausgeführt, sobald man einer Aussage zustimmt, und *reverseWrapUp()*

bei der Ablehnung einer Aussage. Die jeweiligen Funktionen untersuchen dabei die Kind- und Eltern-Knoten eines Arguments und führen während des *wrapUp()* – falls notwendig – einen *todoList_tell()*-Befehl aus. Wird eine Aussage jedoch abgelehnt, führt dies im *reverseWrapUp(),* falls Konsistenz zwischen zwei Argumenten gegeben ist, automatisch zu einem *todoList_untell()*-Befehl. Für den neuen Modus wurden exakt diese Fälle inkorporiert. Sobald einer dieser Fälle einem Nutzer zugeordnet wird, erscheint ein Fenster, das den jeweiligen Fall beschreibt und den Nutzer auffordert, die erwartete Aktion durchzuführen.

3.2 Designentscheidungen und erste Mockups

In einem ersten Schritt wurden für die verschiedenen Fenster Mockups mit dem Programm balsamiq (Balsamiq Studios, o.J.) erstellt. Diese schafften einen ersten Überblick über die Optik der finalen Version. Das Kernziel während der Designentscheidung war, mithilfe möglichst übersichtlicher Fenster für den Nutzer klar darzustellen, welche Schritte als Nächstes erwartet werden. Bekannte Design-Prinzipien bildeten die Basis für die Umsetzung der Fenster. Ein Beispiel dafür ist das 80/20-Prinzip, auch bekannt als Paretoprinzip, das besagt, dass 80% der Ergebnisse mit 20% des Gesamtaufwandes erreicht werden sollen (Koch, 2008, S. 4). Dieses Prinzip, das seinen Ursprung in der Ökonomie hat, findet seinen Weg auch im Design (Lidwell et al., 2010, S. 14). Im Fall von TBDis soll bspw. das Abgeben der Meinung bezüglich einer Aussage mit möglichst wenigen Nutzer-Interaktionen erreicht werden. Wie in Abbildung 3 zu sehen, wurde für den Fall *judge* neu ein fünfstufiges Buttonsystem umgesetzt, das durch einen einfachen Knopfdruck die Meinung des Nutzers annimmt.

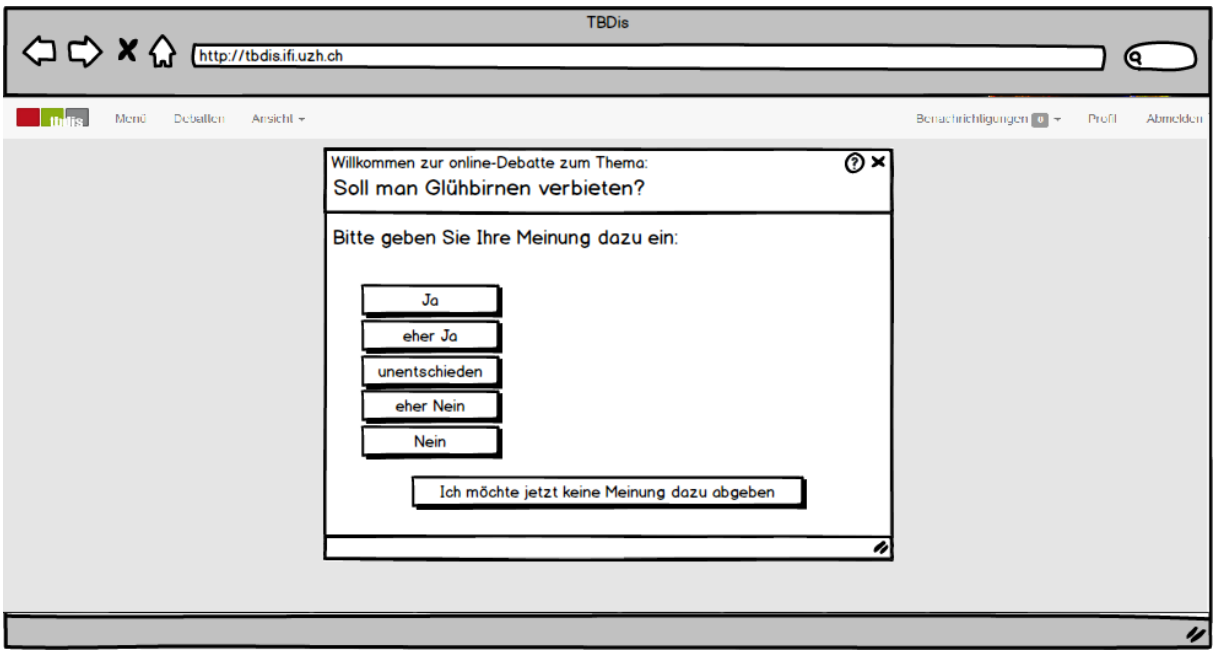

Abbildung 3: Mockup eines Fensters für den *judge*-Fall.

Dieses mehrstufige System war auch in der bisherigen Version präsent. Jedoch wurde vom Benutzer verlangt, zuerst seine einfache Meinung (Ja, Nein oder unentschieden) abzugeben, bevor in einem nächsten Schritt per Schieberegler die Sicherheit über diese Meinung abgefragt wurde. Zu guter Letzt mussten die beiden Werte mit einem zusätzlichen Button abgeschickt werden. Meinungsabgaben im führenden Modus wurden alle so umgesetzt, dass diese drei Schritte in einem zusammengefasst wurden. Durch einfaches Klicken auf den Ja-Button wird bspw. dem Server die Meinung inklusive maximaler Sicherheit übermittelt. Für die Meinungsabgabe wurde bewusst auf ein Buttonsystem gesetzt, da die Applikation – dank der verwendeten bootstrap CSS-Bibliothek („About Bootstrap“, o.J.) – Responsive Web-Design integriert hatte. Responsive Web-Design sorgt dafür, dass die Webseiteninhalte flexibel auf Geräten verschiedenster Auflösungen übersichtlich und benutzerfreundlich dargestellt werden („Was ist Responsive Webdesign, wie funktioniert es?“, Absatz 3). Die Wahl auf den Button-Input fiel, weil damit die optimale Bedienung auf Computern, Tablets und Smartphones gewährleistet ist. Gerade auf Touchscreens spielen die Präzision und Simplizität der Bedienung eine wichtige Rolle. Für die Positionierung der verschiedenen Elemente im Fenster wurde darauf geachtet, dass sie ästhetisch ansprechend ist und eine klare Trennung zwischen unterschiedlichen Aktionen entsteht. Studien haben ergeben, dass Nutzer wohlgestalteten Designs einfachere Bedienbarkeit attestieren, selbst wenn dies nicht der Fall sein sollte (Lidwell et al., 2010, S. 20). So sieht man auch wieder in Abbildung 3, dass im Header des Fensters kurz erläutert wird, um welche Aussage es sich handelt und welcher Art sie ist. Die allseits bekannten Symbole wie das Kreuz zum Schliessen des Fensters und das Hilfesymbol wurden oben rechts in der Ecke umgesetzt. Diese Entscheidung kennt man aus vielen anderen fenster-

basierten Programmen. Damit wird einerseits ein Gefühl der Familiarität beim Nutzer ausgelöst, andererseits lenkt es den Fokus auf das Zentrum des Fensters, wo die Meinung abgegeben wird. Die Gruppierung der Buttons wurde ebenfalls bewusst so gewählt. Das fünfstufige System der Meinungsabgabe wird vom „Ich möchte jetzt keine Meinung dazu abgeben"-Button strikt getrennt. So kriegt der Nutzer das Feedback, dass die zwei Button-Gruppen verschiedene Aktionen durchführen. Das fünfstufige System ist da, um eine Meinung zur Aussage abzugeben und die letzte Option führt dazu, dass keine Meinung abgegeben wird. Positionierung und Gruppierung von Elementen sind ein mächtiges Werkzeug des Designs zur Erstellung von einfach bedienbaren Applikationen. Mithilfe des Designs alleine kann der Nutzer durch die Funktionalität geführt werden.

4 Dokumentation

In diesem Abschnitt wird nachfolgend die Funktionsweise des führenden Modus und TBDis im Allgemeinen dokumentiert. Die Applikation basiert weiterhin auf dem Symfony-Framework und nutzt die bootstrap CSS-Bibliothek. Darauf aufbauend wurde der neue Modus integriert. Den Kern eines Symfony-Projektes bilden die sogenannten Bundles. Sie sind mit Paketen vergleichbar, die man in anderen Applikationen wiederverwenden kann. Ein solches Bundle teilt sich in weitere Subordner auf.

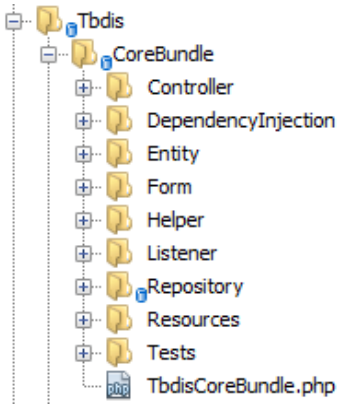

Abbildung 4: Ordnerstruktur des TBDis CoreBundles.

Neben dem TBDis CoreBundle, welches für jegliche Operationen rund um die Debatten zuständig ist, inkorporiert TBDis ebenfalls für die Nutzerverwaltung ein UserBundle, auf das in dieser Arbeit nicht eingegangen wird (für Informationen dazu siehe: „Getting Started With FOSUserBundle “). Nachfolgend wird auf die wichtigsten Komponenten des Projektes eingegangen und erläutert, inwiefern diese für die Applikation relevant sind.

4.1 Controller

Ein Controller ist eine Klasse von PHP-Funktionen, die sich per HTTP-Requests abrufen lassen („Controller (The Symfony Book)“, Absatz 1). Eine solche Funktion wird auch „Action“ genannt und enthält auch im Namen das Anhängsel „Action“, bspw. *todoListTellAction()*. Eine solche Action verlangt bei der Definition eine Annotation, welche die Route des HTTP-Requests definiert und der Route einen Namen gibt. Die Annotation *@Route("/tell", name="todolist_tell")* signalisiert Symfony bspw., dass bei einem HTTP-Request mit den jeweiligen Parametern auf der Route /todolist/tell die *todoListTellAction()* ausgeführt werden

soll. Als Rückgabewert liefert eine solche Action einen HTTP-Response, sei es in Form einer HTML-Datei, einer XML-Datei oder eines serialisierten JSON-Arrays. In den verschiedenen Funktionen können dabei normale PHP-Operationen durchgeführt werden, wie zum Beispiel Datenbankmanipulationen.

4.1.1 DebateController

Der DebateController bietet Funktionen rund um die Debatten selbst, bspw. das Erstellen, Bearbeiten, Löschen oder Darstellen von Debatten. Zu den bestehenden Funktionen des DebateControllers bietet er nun zusätzlich die *showGuidedAction()*. Diese Action wird auf der Route /debate/{id}/guided/show ausgeführt und wird mit dem HTTP-GET-Parameter *id* – der ID der Debatte – aufgerufen. Wird diese Funktion mit einem gültigen *id*-Parameter aufgerufen, wird die von der *showGuided.html.twig* generierte HTML-Datei retourniert, die für die Darstellung der Debatte im führenden Modus zuständig ist.

4.1.2 NodeController

Im NodeController werden Operationen rund um die Knoten des Debattenbaumes vorgenommen. Mit den Funktionen im Controller können neue Knoten beim Erstellen einer neuen Aussage in einer TBDis-Debatte in der Datenbank persistiert oder wieder aus der Datenbank gelöscht werden. Als Rückgabewert werden hier, anders als im DebateController, serialisierte JSON-Arrays retourniert, die Daten des Knotens aus der Datenbank zur Weiterverarbeitung beinhalten. Üblicherweise sind diese Daten bei einer Action die ID des Knotens, der Inhalt des Knotens, der Typ des Knotens (Leitaussage, Zusatzbedingung etc.), der Typ der Aussage (normativ, deskriptiv) und die ID des Elternknotens. Neu wird bei der Bearbeitung eines Knotens der alte Inhalt in die NodeHistory-Tabelle eingetragen.

4.1.3 OpinionController

Hier wird die Meinung in der Datenbank persistiert, die ein Nutzer zu einer Aussage gibt. Die *addOpinionAction()* braucht hierfür als HTTP-POST-Parameter die ID der Debatte und die ID des Knotens. Zusammen mit der ID des Nutzers wird so ein neuer Eintrag in die Opinion-Tabelle der TBDis-Datenbank erstellt.

4.1.4 TodoListController

Der TodoListController bietet das Herzstück der To-do-Liste und inkorporiert die Funktionen des Moderationsalgorithmus. Der Controller ist in erster Linie für das Schreiben, Löschen und Herauslesen von To-do-Listen-Einträgen in der Datenbank zuständig. Alle Funktionen des TodoListControllers brauchen bestimmte HTTP-POST-Parameter, um aufgerufen werden zu können. Beim *_token*-Parameter handelt es sich um ein CSRF-Token, das vor CSRF-Angriffen schützen soll (für Informationen dazu siehe: Kübler, 2014). Nachfolgend werden die verschiedenen Funktionen des Controllers genauer erläutert.

4.1.4.1 todolistTellAction()

Route: /todolist/tell
POST-Parameter: *nodeID, _token, userID, todolistCategory, referenceNodeID(optional)*
Returns: JSON(*nodeid, userid, todolistcategory, action*)

Die *todolistTellAction()* ist angelehnt an Kümins *todolist_tell()*-Funktion seines Moderationsalgorithmus. Die Funktion schreibt einen neuen To-do-Listen-Eintrag in die Datenbank, falls ein solcher noch nicht existiert. Für den Aufruf der Action müssen als HTTP-POST-Parameter die ID des Nutzers bzw. ein Array von IDs der Nutzer übergeben werden, für die ein To-do-Listen-Eintrag erstellt werden soll. Zusätzlich zu den Nutzer-IDs muss die ID des betroffenen Knotens, der To-do-Listen-Fall und falls notwendig die ID des Referenz-Knotens übergeben werden. Der Referenz-Knoten stellt für die Fälle *reconsiderDue2AgreePro, reconsiderDue2AgreeContra* und *reconsiderDue2Inconsistency* den Auslöser-Knoten dar. Anstatt der Nutzer-ID kann auch der String „me“ übergeben werden, der signalisiert, dass der Eintrag für den momentanen Nutzer ausgelöst werden soll. Die Funktion retourniert einen serialisierten JSON-Array, der Informationen über den Typ der Action enthält.

4.1.4.2 todolistUntellAction()

Route: /todolist/untell
POST-Parameter: *nodeID, _token, userID, todolistCategory*
Returns: JSON(*nodeid, userid, todolistcategory, action*)

Das Gegenstück zur *todolistTellAction()* bietet die *todolistUntellAction()*. Basierend auf Kümins *todolist_untell()*-Funktion löscht diese Action einen To-do-Listen-Eintrag aus der Da-

tenbank. Für den Aufruf werden hier fast die gleichen HTTP-POST-Parameter wie bei der *todolistTellAction()* benötigt. Hier entfällt jedoch die optionale ID des Referenz-Knotens, da sie für die Identifikation des Eintrags irrelevant ist. Zusätzlich ist es möglich, statt der tatsächlichen ID des Nutzers den String „all“ zu übergeben. Dies sorgt dafür, dass für alle Nutzer der bestimmte Eintrag gelöscht wird.

4.1.4.3 wrapUpAction()

Route: /todolist/wrapup
POST-Parameter: *nodeID, _token*
Returns: JSON(*nodeid, result*)

Basierend auf Kümins *wrapUp()*-Funktion ist die *wrapUpAction()* entstanden. Diese Action untersucht, ob ein *reconsiderDue2AgreePro-*, *reconsiderDue2AgreeContra-* oder *reconsiderDue2Inconsistency*-Fall für den momentanen Nutzer mithilfe eines *todolist_tell()*-Aufrufs eingetragen werden muss. Aufgerufen wird diese Funktion, sobald ein Nutzer der Leitaussage eines Arguments – egal ob Pro- oder Contra-Argument – zugestimmt hat. In einem ersten Schritt wird untersucht, ob der Nutzer dem gesamten Argument inklusive Zusatzbedingungen zugestimmt hat. Falls dies der Fall ist, wird mithilfe der Helferfunktion *checkConsistency()* überprüft, ob der Nutzer nicht auch einem Argument der gegenteiligen Art mit allen Zusatzbedingungen zugestimmt hat. Hat er dies getan, wird ein *reconsiderDue2Inconsistency* Fall für den momentanen Nutzer in die Datenbank eingetragen. Falls der Nutzer jedoch konsistent geblieben ist, wird abgefragt, ob der Nutzer beim momentanen Knoten nicht einen der anderen beiden Widersprüche ausgelöst hat. Falls der Nutzer sich widerspricht, wird entweder ein *reconsiderDue2AgreePro-* oder *reconsiderDue2AgreeContra*-Eintrag in die Datenbank eingetragen. Anders als bei Kümins *wrapUp()*-Funktion wird in der *wrapUpAction()* neu nicht die ID eines Argument-Knotens erwartet, sondern die ID eines Aussagen-Knotens. Diese Entscheidung wurde getroffen, um die Konsistenz für alle TodoListController-Funktionen zu wahren, da alle Funktionen mit Aussagenknoten-IDs arbeiten.

4.1.4.4 reverseWrapUpAction()

Route: /todolist/reversewrapup
POST-Parameter: *nodeID, _token*
Returns: JSON(*nodeid, result*)

Die *reverseWrapUpAction()* wird bei Ablehnung der Leitaussage eines Arguments ausgeführt. Falls diese Meinung per *checkConsistency()* als konstant deklariert wird, wird der *reconsiderDue2AgreePro-* bzw. *reconsiderDue2AgreeContra*-Eintrag für den Knoten mit einem *todolist_untell()*-Aufruf entfernt. Auch hier wurde die Funktion so umgeschrieben, dass der Aufruf mit der ID eines Aussagenknotens statt der ID eines Argumentknotens aufgerufen wird.

4.1.4.5 retrieveItemWithHighestPriorityAction()

Route: /todolist/getTopItem
POST-Parameter: *treeID, _token*
Returns: JSON(*id, userID, treeID, debateID, actionType, {currentVotes}, {nodeInformation}, {childrenNodeInformation}, {parentNodeInformation}, {grandparentNodeInformation}, {referenceNodeInformation}*)

Die *retrieveItemWithHighestPriorityAction()* wird aufgerufen, wenn Informationen zum obersten To-do-Listen-Eintrag nötig sind. Dabei werden die für die Fenster notwendigen Knoteninformationen aus der Datenbank herausgelesen, bspw. die ID des Knotens, um Operationen darauf ausführen zu können oder aber den Inhalt des Knotens, um ihn dem Nutzer anzeigen zu können. Zusätzlich wird mit der Variable *actionType* der jeweilige To-do-Listen-Fall zurückgeschickt, der es im View-Layer der Applikation ermöglicht, das korrekte Fenster anzuzeigen. Die Funktion bevorzugt den *judge*-Fall, um Nutzer, die noch keine Meinung zur Haupthypothese gegeben haben, aufzufordern, eine Meinung einzugeben. Die restlichen To-do-Listen-Fälle werden normal nach Priorität zurückgeschickt.

4.1.4.6 manageJudgeAction()

Route: /todolist/manageJudge
POST-Parameter: *treeID, _token*
Returns: JSON(*nodeid, userid, todolistcategory, action*)

Die *manageJudgeAction()* wurde für den *judge*-Fall konzipiert. Während im Moderationsalgorithmus von José Kümin der *judge*-Fall lediglich bei der Erstellung der Debatte für alle eingeladenen Nutzer in die Datenbank geschrieben wurde, wird nun dieser Fall stattdessen mit der *manageJudgeAction()* betreut. Diese Entscheidung wurde getroffen, da somit neue Nutzer jederzeit in die Debatte eingefügt werden können. Durch einen einfachen *todolist_tell()*-Aufruf werden nur die Nutzer berücksichtigt, die während der Erstellung der Debatte bereits eingeladen wurden. Würden noch weitere Nutzer zu einem späteren Zeitpunkt eingefügt werden, würde für sie kein *judge*-Fall generiert werden. Diese Action verhindert dies, indem sie für jeden Nutzer sofort ein *judge*-Fall generiert, der sich in die Debatte einklinkt. Sobald er eine Meinung zur Haupthypothese abgibt, wird diese Funktion erneut aufgerufen und der Eintrag wieder aus der To-do-Liste entfernt.

4.1.4.7 manageJudgeArgumentAction()

Route: /todolist/manageJudgeArgument
POST-Parameter: *nodeID, _token*
Returns: JSON(*nodeid, userid, todolistcategory, action*)

Wie für den *judge*-Fall wurde auch für den *judgeArgument*-Fall eine Action erstellt. Die *manageJudgeArgumentAction()* untersucht, ob für einen Leitaussagen-Knoten bereits eine Meinung abgegeben wurde. Anders als bei der *manageJudgeAction()* wird hier jedoch nicht nur für den aufrufenden Nutzer die Meinung untersucht, sondern für alle in der Debatte eingeschriebenen. Sollten also weitere Nutzer zu einem späteren Zeitpunkt zur Debatte zustossen, werden diese beim nächsten Aufruf der Funktion miteinbezogen.

4.1.4.8 manageJudgePremiseAction()

Route: /todolist/manageJudgePremise
POST-Parameter: *nodeID, _token*
Returns: JSON(*nodeid, userid, todolistcategory, action*)

Mit der *manageJudgePremiseAction()* wird der *judgePremise*-Fall betreut. Diese Funktion arbeitet ähnlich wie die *manageJudgeArgumentAction()*, nur dass hier die ID eines Zusatzbedingungs-Knotens statt der ID eines Leitaussagen-Knotens als Input erwartet wird.

4.1.4.9 tellReconsiderDue2EditAction()

Route: /todolist/tellReconsiderDue2Edit
POST-Parameter: *nodeID, _token*
Returns: JSON(*nodeid, userid, todolistcategory, action*)

Die *tellReconsiderDue2EditAction()* wird aufgerufen, sobald ein Nutzer den Inhalt eines Knotens bearbeitet. In einem ersten Schritt untersucht sie für alle Nutzer, ob bereits eine Meinung für diesen Knoten abgegeben wurde. Falls der Nutzer eine Meinung vor der Bearbeitung des Knotens abgegeben hat, wird ein *reconsiderDue2Edit*-Eintrag erstellt. So wird sichergestellt, dass Nutzer keinen To-do-Listen-Eintrag erhalten, die noch keine Meinung zum bearbeiteten Knoten abgegeben haben.

4.1.4.10 manageReconsiderDue2EditAction()

Route: /todolist/manageReconsiderDue2Edit
POST-Parameter: *nodeID, _token*
Returns: JSON(*nodeid, userid, todolistcategory, action*)

Ob ein Nutzer mit einem *reconsiderDue2Edit*-Eintrag seine Meinung geändert hat, wird mit der *manageReconsiderDue2EditAction()* untersucht. Dabei arbeitet sie mit der NodeHistory-Tabelle in der Datenbank und untersucht, ob der Zeitpunkt der Bearbeitung älter als der Zeitpunkt der neuen Meinungsabgabe ist. Ist dies der Fall, wird der Eintrag entfernt.

4.1.4.11 getUpdatedVotesAction()

Route: /todolist/getUpdatedVotes
POST-Parameter: *nodeID, _token*
Returns: JSON(*numberOfProOpinions, numberOfConOpinions, numberOfNoOpinions*)

Die *getUpdatedVotesAction()* zählt alle Meinungen (inklusive noch nicht abgegebenen Meinungen) für den Input-Knoten und retourniert diese in einem serialisierten JSON-Array.

4.1.5 Übrige Controller

Die übrigen Controller (MenuController, NotificationController und OrphanizationController) wurden im Rahmen dieser Arbeit nicht bearbeitet. Der MenuController ist für das Darstellen des Hauptmenüs zuständig. Benachrichtigungen über bearbeitete, gelöschte oder adoptierte Knoten werden im NotificationController verschickt. Der OphanizationController befasst sich mit dem Adoptieren eines Knotens bzw. Unterbaumes.

4.2 Entity

Datenbanken in Symfony können mithilfe des integrierten Doctrine Projects betreut werden. Doctrine setzt sich aus einer Sammlung von PHP-Bibliotheken zusammen, dessen Aufgabe es in erster Linie ist, Daten in Datenbanken zu persistieren und auszulesen („About — Doctrine Project“, Absatz 1). Eine Entity ist eine PHP-Klasse, die lediglich Daten und dessen Zugriffsfunktionen enthält. Der sogenannte Object Relational Mapper (ORM) nimmt bei Durchführung eines Symfony-Konsolenbefehls Entities als Input und konstruiert daraus ein Datenbankschema. So lassen sich mit Doctrine ganze Datenbanken definieren und durch einfache Ausführung eines Befehles erstellen. Durch Annotationen lassen sich Dinge wie Spaltennamen der Tabelle oder Beziehungen zwischen Tabellen definieren.

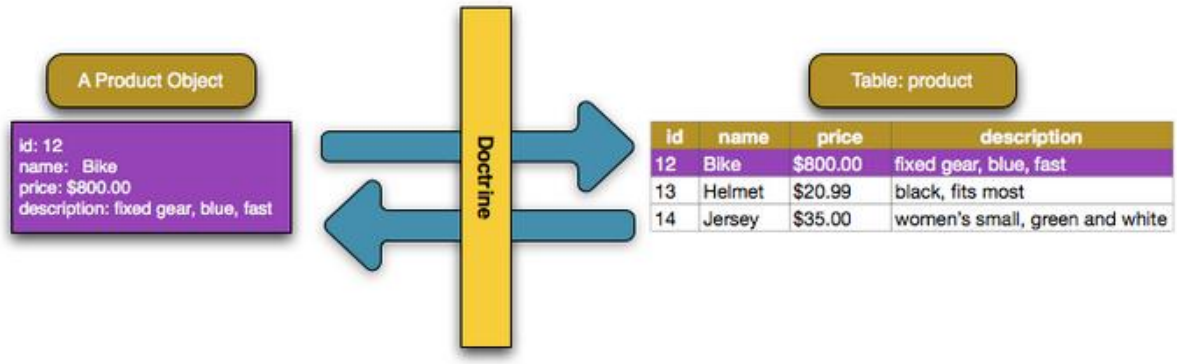

Abbildung 5: Umwandlung einer Entity zu einer Tabelle mit dem Doctrine ORM-Mapper („Databases and Doctrine (The Symfony Book)“, o.J.)

In Symfony wird daher mit solchen Entity Objekten gearbeitet und nicht direkt mit den Tabellen und deren Spalten. Datenbankmanipulationen werden in den Controllern mithilfe des Entity Managers durchgeführt, der dafür zuständig ist, Einträge in die Tabelle zu schreiben oder aber solche zu löschen. TBDis enthält die folgenden Entities: Debate, Node, NodeHistory, Notification, NotficationCenter, Opinion, TodoList, Tree und die im TBDis UserBundle ent-

haltene User-Entity. Nach Durchführung des Symfony-Konsolenbefehls wird dabei das folgende Datenbankschema realisiert:

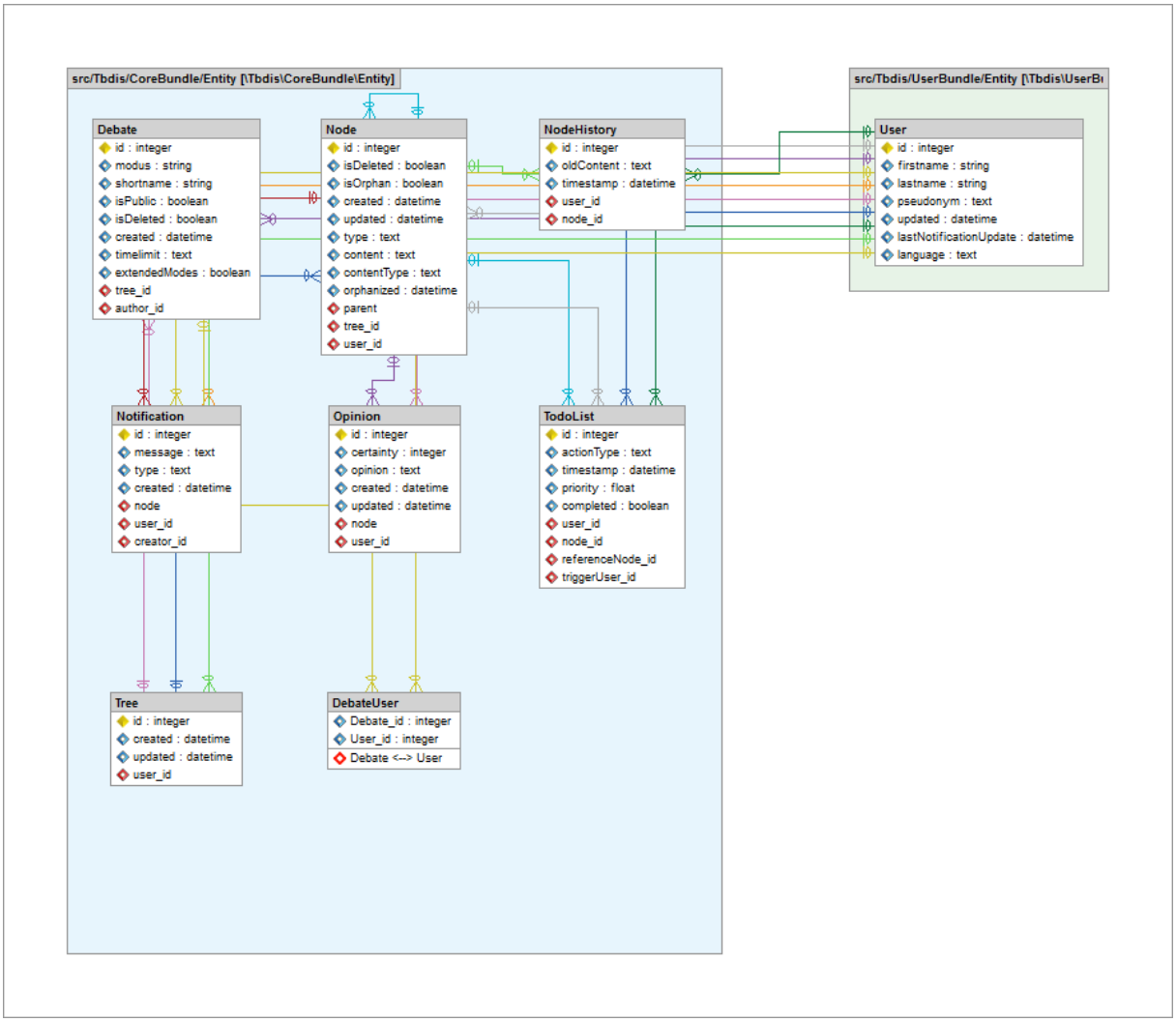

Abbildung 6: TBDis Datenbankschema. Grafik erstellt mit Skipper.

Grundsätzlich orientiert sich das Schema an Küblers Version. In der Debate-Tabelle werden Informationen rund um die Debatte selber abgespeichert. Die Node-Tabelle stellt einen Knoten des Debattenbaumes und dessen Typ dar. Um Notifikationen an Nutzer versenden zu können, werden diese in der Notification-Tabelle abgespeichert. Gibt ein Nutzer eine Meinung zu einem Knoten ab, wird diese in der Opinion-Tabelle persistiert. Die zu einer Debatte eingeladenen Nutzer werden in der DebateUser-Tabelle eingetragen. Informationen zu TBDis-registrierten Nutzern sind in der User-Tabelle wiederzufinden. Der Debattenbaum wird in die Tree-Tabelle geschrieben. Neu wurde die TodoList-Tabelle eingeführt, in der die To-do-Listen-Einträge, deren Fall zusammen mit dem Nutzer und dem betroffenen Knoten gespeichert werden. Zusätzlich wurde die Tabelle NodeHistory erstellt, die während des Bearbeitens des Inhalts eines Knotens den vorherigen Inhalt darin mit einem Zeitstempel abspeichert.

4.3 Repository

Der Entity Manager liefert lediglich die Möglichkeit, Einträge aus Tabellen zu löschen oder zu persistieren. Um Abfragen durchführen zu können, muss mit dem Entity Manager zuerst die Repository aufgerufen werden. Diese liefert einfache Funktionen wie das Auslesen eines Tabellen-Eintrages bezüglich dessen ID (*find()*), das Auslesen der kompletten Tabelle (*findAll()*) oder das Auslesen eines Eintrages bezüglich einfacher Kriterien, wie bspw. dem Typ eines Knotens (*findBy()*). Der Entity Manager erlaubt auch, eigene Repositories zu definieren, die es damit erlauben, komplexere Datenbankabfragen zu konstruieren. Hierfür nutzt Doctrine die eigens definierte DQL (Doctrine Query Language). Getreu den Regeln der objektorientierten Programmierung arbeitet DQL, im Vergleich zum geläufigen SQL (Structured Query Language), mit den als Entity definierten Objekten und nicht mit dem relationalen Schema.

Gewünschte Einträge	**SQL**	**DQL**
Alle Einträge von Node des Nutzers mit der ID 1	SELECT * FROM Node WHERE user_id = 1;	SELECT n FROM TbdisCoreBundle:Node as n WHERE n.user = 1;

Tabelle 3: Vergleich einer SQL und DQL Abfrage

Das obige Beispiel erläutert, wie die verschiedenen Sprachen arbeiten. In SQL werden im *SELECT*-Abschnitt mit „*“ alle Spalten der Tabelle ausgewählt. Im *FROM*-Abschnitt wird mit *Node* die jeweilige Tabelle spezifiziert, bevor in der *WHERE*-Bedingung die Restriktion gesetzt wird, dass die Spalte *user_id* den Wert 1 enthalten muss. In DQL wird die komplette Entity Node, die den Namespace *TbdisCoreBundle:Node* deklarariert hat, selektiert. In der Restriktion wird nicht die Spalte *user_id* berücksichtigt, sondern die Variable *user*, die in der Node-Entity deklariert ist. Die DQL Abfragen werden in Repositories in Funktionen verpackt, sodass im Controller lediglich die Funktion mit dem allenfalls definierten Parameter aufgerufen werden muss. TBDis enthält die folgenden eigens definierten Repositories: DebateRepository, NodeHistoryRepository, NodeRepository, NotificationRepository, OpinionRepository und TodoListRepository.

4.4 Zusammenspiel des Back-Ends

Die bisher erwähnten Komponenten des Symfony-Projektes stellen das Back-End der Applikation dar. Nachfolgend eine Grafik, die das Back-End visuell zusammenfasst.

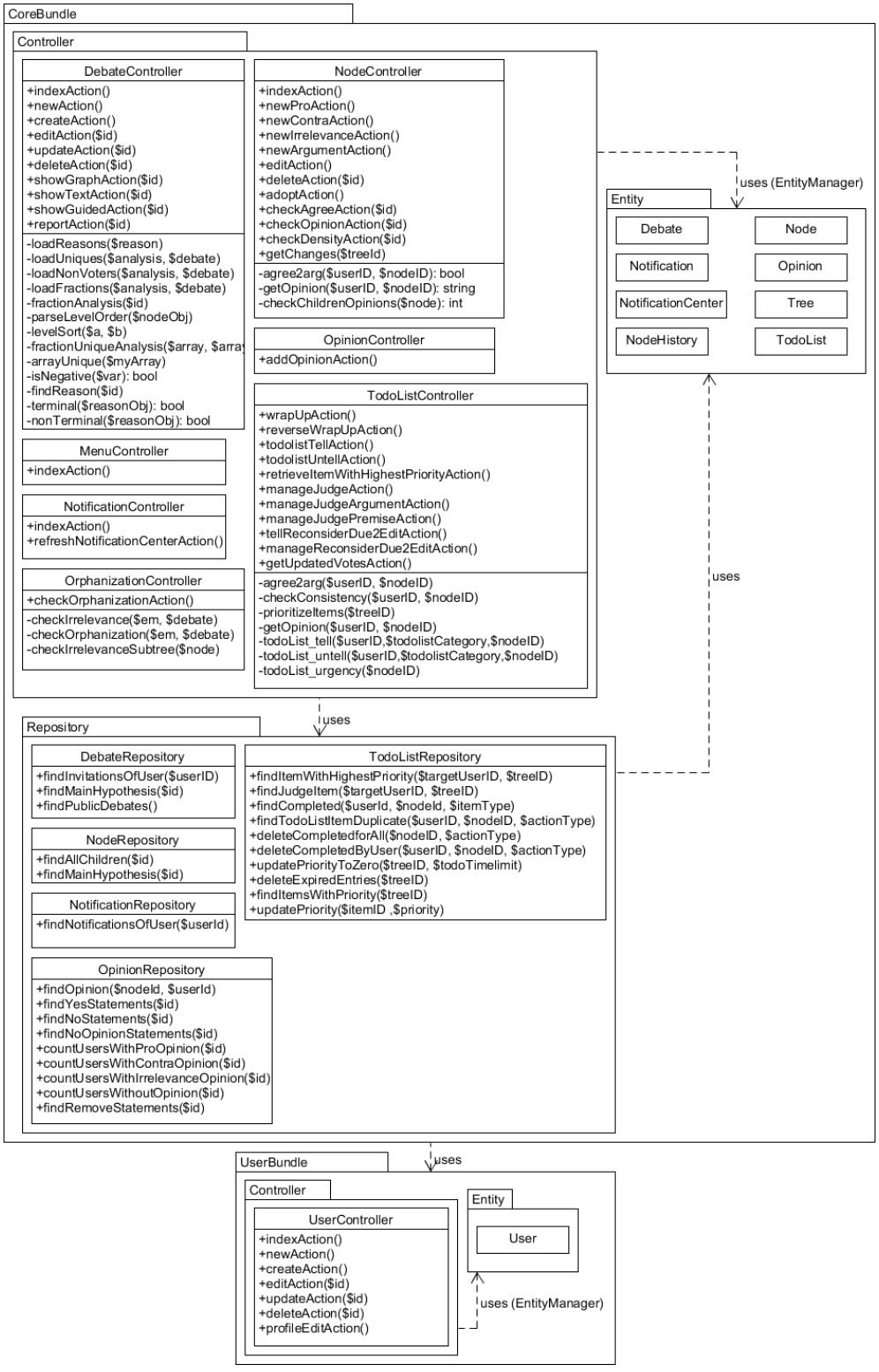

Abbildung 7: Visuelle Darstellung des Back-Ends von TBDis. Erstellt mit UMLet

Zur Architektur lässt sich sagen, dass die Applikation so aufgebaut ist, dass die verschiedenen Controller auch wirklich nur Operationen ausführen, welche die jeweilige Entity betreffen. So kümmert sich der NodeController rein um die Bearbeitung der Knoten. Braucht der Controller einen bestimmten Datenbankeneintrag, so kann er problemlos über die Repositories darauf zugreifen. Diese Trennung der Manipulation der Daten im Controller und dem Herauslesen von Einträgen in der Repository erlaubt, Informationen zu den verschiedensten Tabellen in allen Controllern und vermeidet dadurch Codewiederholungen. Da Funktionalitäten strikt getrennt werden, wird auch das Verstehen des Codes erleichtert. Der ganze Aufbau des Back-Ends greift erneut auf das Thema der modularen Zusammensetzung von Symfony auf. Neue Funktionalitäten lassen sich problemlos durch neue Controller definieren und durch die Repositories lässt es sich problemlos auf alle Entitys zugreifen.

4.5 HTML Twig-Templates

Symfony nutzt die Twig Template Engine (Sensio Labs, o.J.). Damit lassen sich HTML-Dateien generieren, die auch direkt auf Controllervariablen zugreifen können. Mithilfe eines Arrays können Variablen an eine solche html.twig-Datei übergeben werden. Mit zwei geschweiften Klammern wird in der Twig-Datei signalisiert, dass es sich um eine Controllervariable handelt. So wird bspw. mit *{{node}}* auf die in der Action als *node* definierte Variable zugegriffen. Zusätzlich kann auf alle Routen zugegriffen werden, die in den Controllern definiert sind, indem man den zuvor definierten Namen der Route angibt. Mit *{{ path('todolist_tell') }}* zum Beispiel wird der Wert der Route herausgelesen. In diesem Fall wäre dies /todolist/tell. Damit das Template geladen werden kann, muss in einer Controller-Action mit einer Annotation signalisiert werden, welche Datei geladen werden muss. Bspw. wird mit der Annotation *@Template("TbdisCoreBundle:Debate:showGuided.html.twig")* der Action signalisiert, dass beim Aufruf die showGuided.html.twig-Datei geladen werden soll. Wie auch bei üblichen HTML-Dateien können auch bei Twig-Dateien Javascript- bzw. jQuery-Skripte ausgeführt werden.

4.5.1 base.html.twig

Wie der Name schon andeutet, handelt es sich bei der base.html.twig um die Basis aller Templates. In dieser Datei sind Funktionen definiert, die für die verschiedenen Darstellungsmodi von TBDis wiederverwendet werden. Diese Datei wird von keinem Controller direkt

geladen. Sie bietet lediglich Elemente, die an andere Dateien weitergegeben werden. Beispiele hierfür wären Darstellungselemente wie die Navigationsleiste, die in TBDis oben zu finden ist. Auf der Javascript-Seite bietet die base.html.twig verschiedene Funktionen. Für alle Modi bietet es Zugriffsfunktionen für die verschiedenen Dialogfenster, die zur Meinungsannahme geladen werden. Zusätzlich werden auch Funktionen inkorporiert, die zur Bearbeitung des Baumes ausgeführt werden müssen. Die *addNewNode()*-Funktion setzt die richtigen Elemente für das Zeichnen eines Knotens. Wird ein neuer Pro-Knoten benötigt, wird in dieser Funktion bspw. die Farbe auf Grün gesetzt.

4.5.2 showGraph.html.twig

Diese Datei stellt die Debatte in der klassischen Baumstruktur dar. In der *window.onload*-Funktion wird in erster Linie mithilfe von Javascript-Bibliotheken der Baum gezeichnet. Desweiteren sind AJAX (Asynchronous JavaScript and XML) basierte Submit-Funktionen integriert, die ausgelöst werden, sobald der Nutzer ein Formular, wie zum Beispiel das Abgeben einer Meinung zu einem Knoten, ausführt. Diese AJAX-Funktionen schicken die Formulardaten an den angegebenen Pfad der Controllerfunktion und lösen einen POST-Request aus. So wird zum Beispiel beim Abgeben der Meinung die ID des jeweiligen Knotens, die Meinung, die ID der Debatte und die Sicherheit über diese Meinung an /opinion/add versendet. Mit diesen Werten wird die *addOpinionAction()* ausgelöst, welche die neuen Werte in die Datenbank schreibt. Wird diese Anfrage erfolgreich durchgeführt, wird der Baum aktualisiert. Neu werden auch für die To-do-Liste notwendigen Funktionen beim Absenden eines Formulars durchgeführt.

4.5.3 showText.html.twig

Hier wird die Debatte in Textform, um genauer zu sein in einer kaskadierenden Listenform, dargestellt. Im Vergleich zur showGraph.html.twig-Datei wird zusätzlich das jquery.jstree.js-Plug-in geladen, womit der Debattenbaum gezeichnet wird. Neu werden auch hier beim Absenden von Formularen Funktionen zur Bearbeitung der To-do-Liste durchgeführt.

4.5.4 showGuided.html.twig

Der im Rahmen dieser Arbeit entwickelte Modus wird in dieser Datei dargestellt. Im Hintergrund wird der Baum im Graph-Modus geladen, der die exakt gleichen Funktionen wie der

Baum in der showGraph.html.twig besitzt. Somit kann die Debatte in diesem Modus normal weitergeführt werden, selbst wenn der Nutzer gerade keinen vorhandenen To-do-Listen-Eintrag besitzt. In der *window.onload*-Funktion wird im Anschluss des Zeichnens des Baumes die Funktion *manageJudge()* ausgeführt. Diese Funktion führt einen AJAX basierten POST-Request an die *manageJudgeAction()* im TodoListController aus. Nach dem Ausführen dieser Funktion wird die *getNextTodoListmodal()*-Funktion aufgerufen. Dadurch wird zuerst ein Aufruf der *retrieveItemWithHighestPriorityAction()* ausgelöst. Ist dieser erfolgreich, werden alle benötigten Informationen des To-do-Listen-Eintrags retourniert. Je nach To-do-Listen-Fall wird das jeweilige Fenster geladen.

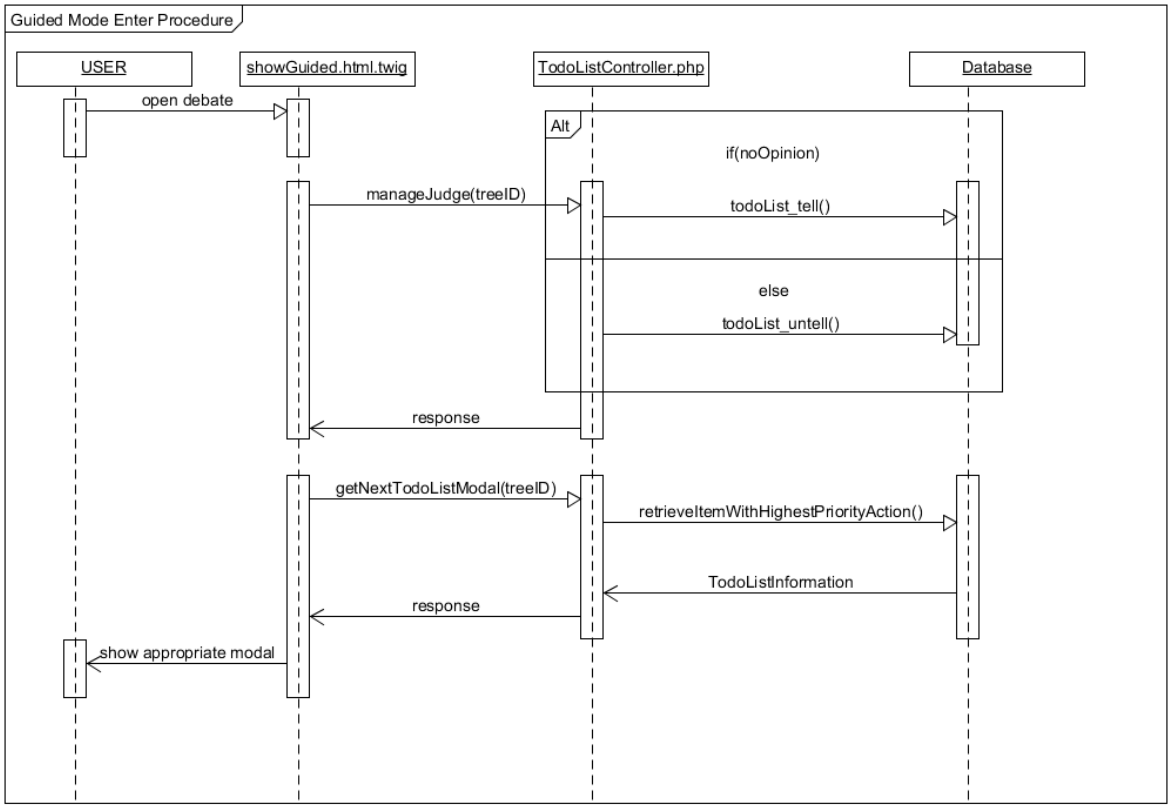

Abbildung 8: Sequenzdiagramm, das die Funktionsaufrufe beim Aufruf der Debatte im führenden Modus schildert. Diagramm erstellt mit UMLet.

In diesem Modus wird zusätzlich nach jeder Aktion des Nutzers, neben den Aufrufen der To-do-Listen-Funktionen, die *getNextTodoListmodal()*-Funktion ausgelöst, die je nach To-do-Listen-Eintrag das jeweilige Dialogfenster aufruft. So wird der Nutzer gleich darauf aufmerksam gemacht, wenn eine Aktion seinerseits notwendig ist. Für das Aufrufen der jeweiligen Fenster wurden Zugriffsfunktionen erstellt, die als Input das To-do-Listen-Objekt annehmen und die jeweiligen Werte für die Variablen in den Fenstern setzen.

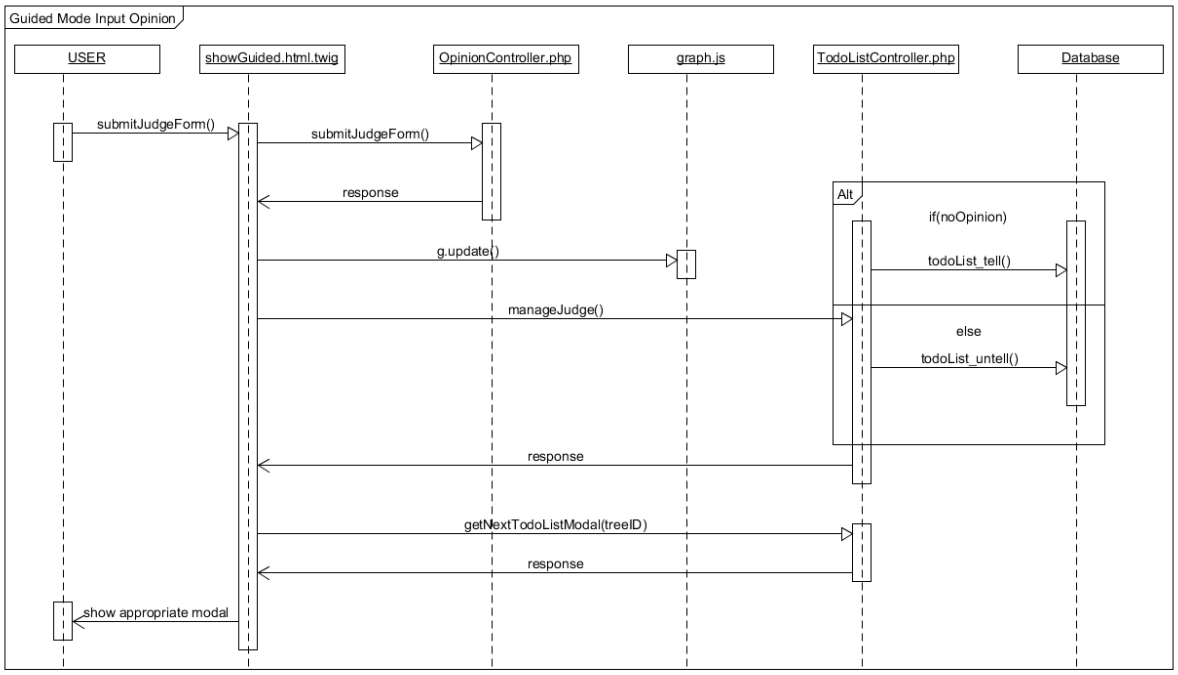

Abbildung 9: Sequenzdiagramm, das die Funktionsaufrufe beim Abgeben einer Meinung zur Haupthypothese aufzeigt. Diagramm erstellt mit UMLet

4.5.5 _modals.html.twig und _guidedModeModals.html.twig

Die Fenster, die sowohl für den führenden Modus als auch für die Meinungsabgabe in den anderen Modi verwendet werden, basieren auf sogenannten bootsrap-Modals. Modals sind Javascript basierte Dialogfenster, die schlichte Funktionalitäten und intelligente Standardwerte bieten („JavaScript Bootstrap“, o.J.). In diesen Dateien sind alle Fenster definiert, die TBDis verwendet. In der _modal.html.twig-Datei befinden sich die Eingabedialoge, die in allen Modi für den Nutzerinput (bspw. Eingabe der Meinung oder Erstellung eines neuen Knotens) verwendet werden. Die Datei _guidedModeModals.html.twig enthält zusätzlich die für den führenden Modus notwendigen Fenster. Gekennzeichnet durch das HTML-Attribut *id* wird in den Zugriffsfunktionen mit jQuery-Selektoren auf die jeweiligen Fenster zugegriffen und mit dem Befehl *show()* bzw. *hide()* das jeweilige Modal ein- und ausgeblendet. In den Fenstern werden übliche bootstrap-Elemente verwendet, wie Buttons oder Textfelder. Wird bei Fenstern mit nur einer Option auf einen Button geklickt, löst dies einen Submit aus, der den Aufruf auf die jeweilige Route per AJAX-POST-Request absendet. Bei Fenstern mit mehreren Optionen wird eine Funktion ausgelöst, die erst einen AJAX-POST-Request auf die Route schickt. Dies wurde so umgesetzt, da für Fenster mit mehreren Optionen auch mehrere Formulare hätten definiert werden müssen. Dies hätte in erster Linie die Datei mit den Modals

unnötig aufgebläht, andererseits hätte dann auch zusätzlich für jedes Formular noch ein Submit geschrieben werden müssen, was den Code unleserlicher gemacht hätte. Der Nachteil, der dadurch entstehen könnte, ist, dass man ohne Submit nicht die Möglichkeit hat, seine Meinung mit der Return-Taste einzugeben. Dies stellt aber bei Fenstern mit mehreren Optionen ohnehin kein Problem dar und bei Fenstern mit nur einer Option wurde der Submit belassen, sodass weiterhin mit der Return-Taste die Meinung bestätigt werden kann.

4.6 Sprachdateien

Symfony inkorporiert einen sogenannten Translator, mithilfe dessen man Texte auf eine externe YAML-Datei auslagern kann. YAML („YAML™ Version 1.2“, o.J.) verwendet eine hierarchische Strukturierung der Variablen, die durch Einzüge definiert wird. Dies ermöglicht es, Texte in verschiedenen Sprachen zu definieren und diese dann in das Twig-Template als Variable zu laden. Diese lassen sich, ähnlich wie die Controllervariablen, mit den doppelt geschweiften Klammern und dem Zusatz „trans“ aufrufen. Für den Willkommenstext des *judge*-Modals muss dem Twig-Template bspw. Folgendes übergeben werden: {{ 'tbdis.guided.judgeModal.welcome'|trans }}. Alle Texte in TBDis wurden in Sprachdateien ausgelagert, was ermöglicht, die Texte leicht anzupassen und neue Sprachen einzuführen, da nur eine weitere Datei mit dem gleichen Schema erstellt werden muss.

4.7 Javascript-Bibliotheken

Da sich diese Arbeit nicht mit der Darstellung des Baumes befasste, wurden die Javascript-Bibliotheken unverändert gelassen. Um die Dokumentation jedoch vollständig zu halten, wird an dieser Stelle kurz erwähnt, welche Rolle die verschiedenen Klassen spielen. Die Bibliothek graph.js ist für das Zeichnen der Knoten und Kanten des Graphenbaumes der Debatte zuständig. Hierfür werden mithilfe von raphael.js, die es erlaubt, Vektorgrafiken zu erstellen, die zu zeichnenden Elemente erstellt. Die Elemente werden dann auf einer definierten Leinwand gerendert. Mit X- und Y-Koordinaten wird die Position der Elemente auf der Leinwand bestimmt. Die graffle.js-Bibliothek ist für das Erstellen der Kanten zuständig. Mithilfe von jquery.jstree.js wird die Debatte in Textform dargestellt. CANDISLib.js bietet AJAX-Funktionen, welche die Aktualisierung des Baumes auslöst, ohne dass eine Aktualisierung der Seite notwendig ist. Zu guter Letzt bietet CANDISHelper.js Hilfsfunktionen wie bspw. die

HTML-Enkodierung eines Strings für AJAX-Aufrufe oder das Vorbereiten des POST-Request-Strings.

4.8 Zusammenspiel des Front-Ends

Mit dem Ende des vorangehenden Abschnittes wurde auch das Front-End so weit erläutert. Die folgende Abbildung fasst das Front-End zusammen.

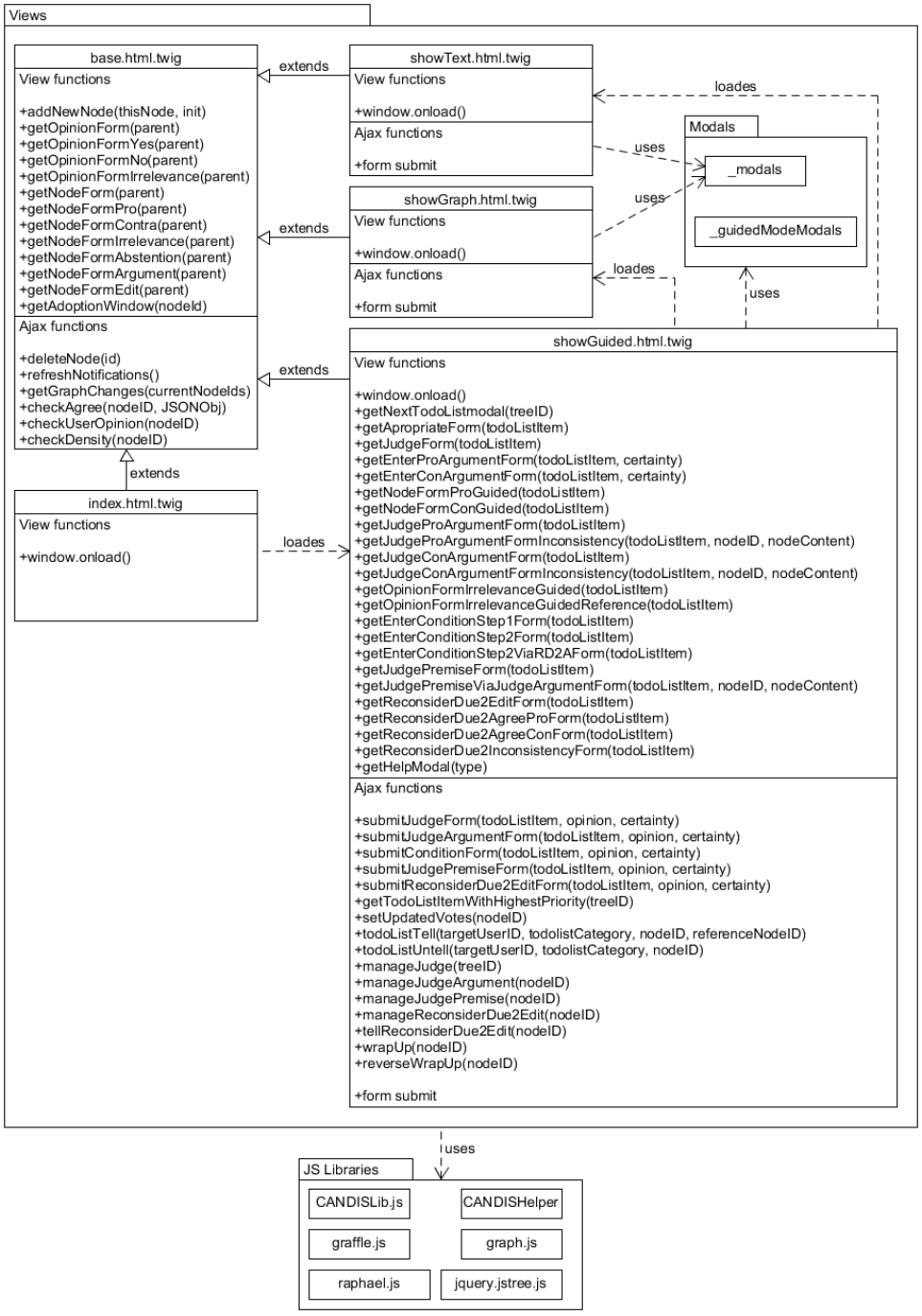

Abbildung 10: Visuelle Darstellung des Front-Ends von TBDis. Erstellt mit UMLet

Auch hier hält sich das Motto von Symfony quer durch die Architektur. Die base.html.twig bietet die wichtigsten Funktionen, die für einen Darstellungsmodus notwendig sind. Die Dateien, die davon erben, erhalten zusätzliche Funktionen, die den jeweiligen Modus erweitern. Hierbei fällt die Datei für den führenden Modus auf, die für alle verschiedenen Dialogfenster eine Zugriffsfunktion und eine Submit-Funktion benötigt. Die verschiedenen Dialogfenster werden in separate Dateien ausgelagert, die es ermöglichen, weitere Fenster zu definieren. Die Darstellung der Debatte wird in den verschiedenen Javascript-Bibliotheken betreut und in den html.twig-Dateien ausgeführt.

4.9 Bugs in TBDis

Während der Entwicklung und des Testens sind einige bestehende Bugs in der TBDis-Applikation festgestellt worden. Wird eine Debatte öffentlich gemacht, so führt dies dazu, dass die Teilnehmer keine Möglichkeit mehr haben, an der Debatte teilzunehmen. Alle Interaktionsmöglichkeiten werden ausgeblendet und der Nutzer wird wie ein Gast behandelt. Folglich wird auch der führende Modus nicht ausgelöst und es erscheinen keine Dialogfenster. Des Weiteren nimmt bei der Debattenerstellung das Feld „Zeitlimit“ andere Werte als Zahlen an, was potenziell zu Abstürzen der Applikation, die durch Komponente, die auf das Zeitlimit angewiesen sind, führen könnte. Ein weiterer Darstellungsbug entsteht beim Eingeben von falschen Log-in Daten: Anstatt eine Meldung auszugeben, die für einen Nutzer ohne IT-Kenntnisse verständlich ist, wird lediglich ein Text angezeigt, der durch die Exception ausgelöst wird. Diese Bugs sind zwar für die Nutzung von TBDis nicht allzu kritisch, jedoch müssen sie vor einer Live-Schaltung der Applikation ausgemerzt werden.

5 Fazit

Mit der Implementation des führenden Modus wird eine fundierte Basis geschaffen, TBDis dem Endnutzer näher zu bringen. Indem der Nutzer mit den Dialogfenstern durch die erwarteten Aktionen geführt wird, erhält er Hilfestellungen, die ihm bei der Teilnahme an einer TBDis-Debatte assistieren und helfen, seine Meinung zu den Diskussionspunkten korrekt abzugeben. Der Fokus wird dabei auf die Punkte gesetzt, die möglichst früh zu einem Konsens bzw. Dissens führen. Erfahrenere Nutzer haben nichtsdestotrotz die Möglichkeit, jederzeit aus dem führenden Modus auszusteigen und frei ohne Einschränkungen die Debatte nach ihren Vorlieben zu führen. Durch das ständige Aktualisieren der To-do-Liste im Hintergrund, auch in anderen Modi, kann man bei Ratlosigkeit jederzeit wieder in den führenden Modus einsteigen. Während dieser Arbeit wurde jedoch klar, dass selbst dieser Modus ein gewisses Basiswissen über Aussagenlogik voraussetzt, um vollständig nachvollziehen zu können, welche Auswirkungen die verschiedenen Handlungsoptionen auf die eigentliche Debatte haben. Beispielsweise erhält der Nutzer kein wirkliches Feedback, was das Einführen einer zusätzlichen Bedingung zu einem Argument bewirkt. Ein weiteres Problem stellt die reine dialogfensterbasierte Debattenführung dar. Es hat sich zwar herausgestellt, dass diese noch am Anfang der Debatte sehr gut funktioniert, jedoch beim tieferen Eindringen in den Debattenbaum zu Übersichtsproblemen führt. Kurz gesagt: Je höher die Komplexität des Debattenbaumes, desto mehr verliert man die Orientierung, um welchen Knoten man in dem Moment gerade urteilt. Zwar hat man jederzeit die Möglichkeit, das Dialogfenster zu schliessen und sich somit den Überblick über die Debatte zu verschaffen, jedoch führt dies zu Umständlichkeiten in der Bedienung.

6 Ausblick

Software befindet sich nie in einem finalen Stadium und steht im stetigen Wandel. Diese Aussage betonen Birrel und Ould (1988) in Ihrem Handbuch zur Softwareentwicklung. Davon bleibt TBDis natürlich nicht verschont. Während nun die Basis für freundlichere Bedienung mithilfe des führenden Modus geschaffen wurde, bietet dieser und TBDis im Allgemeinen grosses Potenzial, sich noch weiter zu entwickeln. So könnte zukünftig für die bessere Erläuterung der Aussagenlogik ein weiterer Modus eingeführt werden, der den Nutzer in der Form eines Tutorials durch eine Beispieldebatte führt. Dabei können die verschiedenen Elemente erläutert werden, indem der Nutzer eine Debatte hautnah miterlebt. Für den führenden Modus bieten sich auch noch einige Verbesserungen an. Um das Problem der Übersicht zu lösen, könnte man die bis jetzt dialogfensterbasierte Implementierung so umstrukturieren, dass der Debattenbaum trotzdem ständig im Blick ist. Den momentan zu behandelnden Knoten könnte man dann im Debattenbaum kennzeichnen, sodass gleich ersichtlich ist, wie tief man sich im Debattenbaum befindet. Eine weitere Optimierungsmöglichkeit wäre, zukünftig die komplette To-do-Liste an den Nutzer zu verschicken, sodass er sich aussuchen kann, welchen Eintrag er als Nächstes behandeln will.

7 Quellenverzeichnis

About Bootstrap. (o.J.). Aufgerufen am 04.08. 2015, unter http://getbootstrap.com/about/

About — Doctrine Project. (o.J.). Aufgerufen am 04.08. 2015, unter http://www.doctrine-project.org/about.html

Balsamiq Studios. (o.J.). Balsamiq. Rapid, effective and fun wireframing software. - Balsamiq. Aufgerufen am 06.08. 2015, unter https://balsamiq.com/

Birrell, N. D., & Ould, M. A. (1988). *A practical handbook for software development*. Cambridge [England]; New York: Cambridge University Press.

Burstein, F. (2008). *Handbook on decision support systems 1: the basics* (1st ed). New York: Springer.

Controller (The Symfony Book). (o.J.). Aufgerufen am 04.08. 2015, unter http://symfony.com/doc/current/book/controller.html

Databases and Doctrine (The Symfony Book). (o.J.). Aufgerufen am 04.08. 2015, unter http://symfony.com/doc/current/book/doctrine.html

Fjermestad, J., & Hiltz, S. R. (2000). Group Support Systems: A Descriptive Evaluation of Case and Field Studies. *Journal of Management Information Systems*, *17*(3), 115–159.

Getting Started With FOSUserBundle (The Symfony Bundles Documentation). (o.J.). Aufgerufen am 04.08. 2015, unter https://symfony.com/doc/master/bundles/FOSUserBundle/index.html

Hilty, L. (2010). Projekt CANDis – Computer-Aided Normative Discourse Kurzbeschreibung.

Hilty, L. (2011). CANDis-Moderationsalgorithmus V2.3.

JavaScript Bootstrap. (o.J.). Aufgerufen am 07.08. 2015, unter http://getbootstrap.com/javascript/

Johansen, R. (1991). Groupware: Future directions and wild cards. *Journal of Organizational Computing, 1*(2), 219–227. http://doi.org/10.1080/10919399109540160

Koch, R. (2008). *The 80/20 principle the secret of achieving more with less*. New York: Doubleday. Heruntergeladen unter http://www.contentreserve.com/TitleInfo.asp?ID={23FE754F-3BFE-4616-9FD7-779795AE7AE8}&Format=50

Kübler, N. (2014). Portierung und Weiterentwicklung von TBDis auf Symfony mit dem Schwerpunkt Sicherheit.

Kümin, J. (2010). Prototyp einer Webapplikation zur Unterstützung der Strukturierung von Debatten.

Kümin, J. (2012). Evaluation of Multivariate Methods for Dimensionality Reduction and Cluster Analysis in the Context of a Debate Structuring System.

Lidwell, W., Holden, K., & Butler, J. (2010). *Universal principles of design: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design ; [25 additional design principles]* (rev. and updated). Beverly, Mass: Rockport Publ.

Mason, R. O., & Mitroff, I. I. (1981). *Challenging strategic planning assumptions: theory, cases, and techniques*. New York: Wiley.

Rautenberg, W. (1979). *Klassische und nichtklassische Aussagenlogik*. Braunschweig; Wiesbaden: Vieweg. Heruntergeladen unter http://catalog.hathitrust.org/api/volumes/oclc/6224781.html

Sensio Labs. (o.J.). About - Twig - The flexible, fast, and secure PHP template engine. Aufgerufen am 04.08. 2015, unter http://twig.sensiolabs.org/

TBDis Help-Wiki. (2012). TBDis Help-Wiki. Aufgerufen am 03.08. 2015, unter http://tbdis.org/wiki/index.php/Help_DE

Thurlow, C., Lengel, L., & Tomic, A. (2011). *Computer mediated communication: social interaction and the internet* (Reprinted). Los Angeles: SAGE.

Was ist Responsive Webdesign, wie funktioniert es? | wendweb. (o.J.). Aufgerufen am 04.08.2015, unter http://www.responsive-webdesign.mobi/was-ist-responsive-webdesign/

YAML Ain't Markup Language (YAML™) Version 1.2. (o.J.). Aufgerufen am 08.08. 2015, unter http://www.yaml.org/spec/1.2/spec.html

8 Anhang

8.1 Installation des Symfony-Projektes

Für die Installation von TBDis sind einige Vorbereitungsschritte notwendig. Der Server, auf dem TBDis installiert werden soll, benötigt einen Apache Server, PHP Version 5.5+ und MySQL.

1. TBDis-Dateien auf den Server kopieren
2. Im Virtualhost, als DocumentRoot den Ordner *web* setzen.
3. In der Datei *app/config/parameters.yaml* die Datenbankinformationen aktualisieren. (Port, Passwort, Datenbankname und Datenbanknutzer).
4. Im root-Verzeichnis des TBDis-Projektes folgende Befehle ausführen:
    a. php composer.phar self-update
    b. php composer.phar update
    c. php app/console doctrine:database:create
    d. php app/console doctrine:schema:update --force
    e. php app/console fos:user:create admin --super-admin

Anschliessend wird man für den Administrator nach einem Passwort und einer E-Mail-Adresse gefragt. Hat man diese erfolgreich eingegeben, kann man sich als Administrator bei TBDis einloggen.

8.2 Versionierung mit git

Für dieses Projekt wurde ein git Repository erstellt, welches es vereinfachen soll, zukünftig den Code zu warten und zu verteilen. In diesem Abschnitt werden die wichtigsten Befehle und ein typischer Arbeitsablauf erläutert.

Beschreibung	**Befehl**
Git repository clonen (herunterladen)	git clone <Link des repositorys>
Veränderte Dateien anzeigen	git status
Veränderungen aus dem Repository holen	git pull
Alle Dateien zum Commit hinzufügen	git add .
Bestimmte Datei zum Commit hinzufügen	git add <Verzeichnis der Datei>
Einen Commit tätigen	git commit –m „<Beschreibung des Commits>“
Commit pushen	git push

Tabelle 4: Die wichtigsten Befehle für die Handhabung eines git Repositorys

Um das Projekt aus dem Repository auf die lokale Festplatte zu laden, muss zuerst ein Clone-Befehl getätigt werden. Werden Veränderungen im Projekt getätigt, empfiehlt es sich, zuerst mit „git status“ die veränderten Dateien anzusehen. Anschliessend sollte man mit einem Pull-Befehl alle Änderungen aus dem Repository holen, falls andere Personen gleichzeitig am Projekt arbeiten. Somit können Merge-Konflikte frühzeitig ausgemerzt werden. Nach dem Pull fügt man die Dateien mit dem Add-Befehl zu seinem nächsten Commit hinzu, bevor man dann den eigentlichen Commit tätigt. Hier hat man mit dem Modifier „-m“ die Möglichkeit den Commit kurz in ein paar Worten zu beschreiben. Abschliessend werden die Commits mit einem Push-Befehl in das online Repository veröffentlicht.