# Projektbericht “tudo”

Vanessa Speeth

Matrikelnr. 192969
vanessa.speeth@tu-dortmund.de

Merlin Scholz

Matrikelnr. 192505
merlin.scholz@tu-dortmund.de

2018-07-15

# Contents

# 1 Erste Planungen

Die erste Überlegung beim Erhalt der Projektvoraussetzungen bestand, wie bei den meisten Gruppen, zuerst aus den Fragen: "Wie soll unsere Seite aussehen?" und "Wie soll sie funktionieren?". Wir haben einen relativ allgemeinen Ansatz für den Typ unserer Webapplikation gewählt. Während in den Projektvoraussetzungen zum Beispiel von einem Issue-Tracker oder einem Wochenend-Trip-Planer gesprochen wird, setzten wir uns das Ziel, eine allgemeine, für viele Einsatzbereiche geeignete, funktionale und für Poweruser geeignete ToDo-Liste zu erstellen. Dieser Plan beinhaltet eine Übersicht über die wichtigsten Einträge, diverse Filterfunktionen und teilweise auch eine Analysefunktion.

# 2 Design

Die Gestaltung des Designs war einer der wichtigsten Punkte in der Erstellung unserer Webapplikation. Wie kann man viele Informationen, viele Einträge kompakt, und trotzdem ordentlich und überschaubar auf einer Seite darstellen? Für diesen Aspekt haben wir uns in Konzept Designs von anderen Anwendungsarten eingearbeitet, die das selbe Ziel erfüllen müssen: Email-Clients, die eine Liste von vielen einzelnen Emails anzeigen müssen, und Banking-Websites, die kompakt die letzten Überweisungen anzeigen müssen. Als erstes Konzept hat unsere Gruppe auf ein drei-Spalten-Design geeinigt: Von links nach rechts, eine Spalte für Kategorien, in die man die Einträge aufteilen konnte, eine Spalte für die Liste der eigentlichen Aufgaben, und eine Spalte für die Details der etwaigen markierten Aufgabe. Nachdem wir bereits mit der Implementierung dieses Designs angefangen haben, d.h. die entsprechende Datenstruktur erstellt, die REST-APIs und Angular-Seite inklusive Design und Anfragen ist uns aufgefallen, dass dieser Ansatz noch nicht optimal ist. Wie sollte der Nutzer alle Aufgaben (ohne die Kategorien, also ungefiltert) auf einen Blick sehen? Geht durch dieses drei-Spalten-Design nicht zu viel horizontaler Platz verloren? Was wenn eine Aufgabe in keine oder mehrere Kategorien passt? Auf diese Fragen konnte unser erstes Design keine Antwort geben, es ging also zurück in die Design-Planung. Unsere zweite Idee lässt sich mit einem einfachen Wort beschreiben: Tags. Wir schufen die Kategorien komplett ab, und ermöglichten es dem User, seinen Aufgaben einzelne Tags zuzuordnen. Die Funktionalität ging bei diesem Ansatz auch nicht verloren, da der Nutzer einfach nach Tags filtern kann.

Abgesehen von unserer Aufgabenliste, die als wichtigstes Element einen Großteil der Seite einnimmt, enthält unsere Webapplikation einen Header mit unserem Logo, der Suchleiste und einigen grundlegenden Links (wie zum Beispiel die Accountverwaltung oder um sich abzumelden). Darunter befindet sich eine zweite horizontale Leiste, um sämtliche Tags anzuzeigen. Dieses Element beinhaltet auch die Funktionen zum Tag anlegen und löschen. Das linke Drittel von tudo bietet weitere Filter, um den Zeitraum einzugrenzen und um zwischen eigenen und zugewiesenen Items zu wechseln. Des Weiteren lassen einzelne Items sich "aufklappen", dort gibt es Funktionen, um die jeweiligen Eigenschaften, wie zum Beispiel die Tags, zu ändern. Auf der Anmeldung- (und Registrierungs-) Seite war das Hauptziel, die relativ wenigen UX-Elemente auf die große "Leinwand" zu bringen, ohne dass die Seite zu leer wirkt. Dies haben wir dadurch erzielt, dass auf dem rechten Drittel der Seite das entsprechende Anmelde- oder Registrierformular weiß hinterlegt ist, während die linken zwei Drittel von einem Stock-Foto geziert werden. Dieses Foto hatte keinen besonderen Nutzen, außer dass es nicht zu ablenkend von den Formularen sein sollte und farblich zum Rest der Seite passen sollte.

Das eigentliche UI (statt dem eben beschriebenen UX) ist eher einfach und farblos gehalten, damit es nicht zu sehr von der eigentlichen Funktionalität ablenkt, es existiert allerdings grün als Akzentfarbe.

# 3 Maven Aufbau

Bei dem Maven Projektaufbau haben wir uns anfangs stark nach den Beispielprojekten gerichtet, da dieser Ansatz für uns optimal schien. So haben wir eine Haupt-`pom.xml`, in der drei Module definiert sind: Eins für JPA, eins für die API, und eins für das Frontend. Der Vorteil an dieser Einteilung besteht darin, dass man bei der Entwicklung den Code in die drei MVC-Kategorien einteilen kann: Model (Daten), View (Frontend)

und Controller (API). Die einzelnen Module waren stufenweise voneinander abhängig, mit *tudo-jpa* ohne projektinterne Abhängigkeiten und *tudo-web* mit allen Modulen als (indirekte) Abhängigkeit, und haben alle tudo als Parent-Projekt.

Dieser Aufbau hatte in der Praxis allerdings einige Nachteile, so waren zum einen Konfiguratonsdateien unübersichtlich auf die Module verteilt, andererseits führte der Ansatz zu sehr langen Kompilierungszeiten, ohne dabei zu, in der Praxis relevanten und signifikanten, Vorteilen zu führen. Die lange Kompilierungsdauer führten wir darauf zurück, dass wir, wenn wir auf mehreren Modulen Änderungen durchgeführt hatten, wir erst auf dem Parent-Modul `mvn clean install` und dann auf *tudo-web* das entsprechende Kommando, um die `.war`-Datei zu erstellen, ausführen mussten, statt alles in einem. Dies führte auf unserem Entwicklungsserver zu Kompilierungszeiten von bis zu 5 Minuten. Nachdem in etwa der Mitte dieses Projektes diese Fehler zu gravierend wurden und somit das Projekt zu unübersichtlich, haben wir uns entschieden, alles in einem einzigen Parent-Artifact zu vereinen. Dies ermöglichte uns auch wesentlich einfacher auszuwählen, ob AngularDart mit kompiliert werden sollte oder nicht, das Dart Plugin wird so einfach über ein standardmäßig aktiviertes Profil eingebunden. Bei dem alten Ansatz war dies komplizierter, da wir ein leeres *tudo-web*-Modul hätten erzeugen müssen. Die Erstellung von Profilen wird durch diesen Ansatz im Allgemeinen wesentlich vereinfacht, da dies früher nur auf dem *tudo-web*-Modul möglich war.

Wie eben erwähnt enthält unser Maven Projekt nur ein einziges, standardmäßig aktiviertes Profil, namentlich `buildng`. Dies dient dazu, beim Projektbau automatisch das Dart Projekt zu compilen. Erreicht wird dieser build-Schritt über das Maven Plugin `com.github.dzwicker.dart-maven-plugin`. Ein weiteres, nicht standardmäßig installiertes Plugin in unserer Webapplikation ist das `org.wildfly.plugins.wildfly-maven-plugin`-Plugin. Dies erleichterte uns den build-Prozess wesentlich, da wir durch das Ziel `wildfly:redeploy` unser verändertes Projekt sehr einfach in wenigen Sekunden auf einen laufenden WildFly Server neu übertragen konnten, ohne selbst die `.war`-Datei zu erstellen und eigenhändig hochzuladen.

# 4 Projekt- und Java-Aufbau

Auf der Projektebene existiert, wie in einem üblichen Projekt, hauptsächlich ein `src` Ordner. In dem darin beinhalteten `main` Ordner befinden sich folgende Unterordner:

- dart

  Hier befinden sich die Dart Quelldateien und Bibliotheken, also hauptsächlich unser AngularDart Projekt und seine Abhängigkeiten. Beim Maven build-Prozess wird hier, wie beschrieben, automatisch `pub get` und `pub build` aufgerufen. Die generierten Dateien werden im Ordner `src/main/webapp/web` abgelegt.

- java

  In diesem Ordner befindet sich der komplette Java-Quellcode. Es existieren folgende Packages in ihren jeweiligen Ordnern:

  - `de.webtech2.tudo.auth` Hier befinden sich die für Apache Shiro notwendigen Realms und der `RestServiceAuthenticator`
  - `de.webtech2.tudo.conf` Hier befinden sich der `JacksonProvider` und der `JAXRSActivator`.
  - `de.webtech2.tudo.dao` Hier befinden sich die Data Access Objects, also die Klassen, über die unser Projekt auf die Datenbank zugreift. Alle DAOs erben von der abstrakten Klasse `GenericDAO` und werden durch den `DAOCreator` instanziiert.

    * `de.webtech2.tudo.dao.requestBuilder` In diesem Package liegt eine Klasse, die es uns nach dem Builder-Prinzip ermöglicht, eine Liste von todo-Items nach speziellen Filterkriterien anzufordern.
  - `de.webtech2.tudo.model` Hier liegen die drei Model-Dateien (User, Item und Tag). Diese Klassen haben Hibernate Annotiationen.
  - `de.webtech2.tudo.rest` Hier befindet sich nur die `REST.java`, die für alle REST-Anfragen zuständig ist. Am Anfang dieses Projektes war die REST API pro Pfad in eine seperate Datei aufgegliedert, allerdings ist uns bei der Entwicklung aufgefallen, dass alle Anfragen mit dem Pfad `/users/` beginnen.

- resources

  In diesem Ordner befinden sich in den Unterordnern `META-INF` und `WEB-INF` die Konfigurationsdateien für Java Beans und für die persistence-Schnittstelle.

- webapp

  Dieser Ordner beinhaltet die Konfigurationsdateien für die Java EE-Webschnittstelle (in den jeweiligen Unterordnern), die Webseiten für das Account-Management (Anmeldung, Registrierung), und einige typische Dateien wie die favicons. Des Weiteren wird, wie vorher beschrieben, das AngularDart-Projekt hier in den Order `/web/` kompiliert.

# 5 REST-APIs

Die REST-APIs wurden, wie nach Projektvorgabe, durch JAX-RS implementiert. Initialisiert wird die entsprechende Klasse durch `de.webtech2.tudo.conf.JAXRSActivator`. Es existieren folgende URLs mit den entsprechenden Methoden:

```
http://0.0.0.0/tudo/api/users/
GET, POST

http://0.0.0.0/tudo/api/users/{username}
HEAD, GET, PUT, DELETE

http://0.0.0.0/tudo/api/users/{username}/items
GET, POST

http://0.0.0.0/tudo/api/users/{username}/items/{id}
PUT, GET, DELETE

http://0.0.0.0/tudo/api/users/{username}/tags
GET, POST

http://0.0.0.0/tudo/api/users/{username}/tags/{id}
PUT, GET, DELETE
```

Bei der individuellen Benutzer-Abfrage ist die HEAD-Methode seperat aufgeführt, da sie weniger Berechtigungen benötigt, als die gewöhnliche GET Anfrage. Dies rührt daher, dass die HEAD-Methode auf der Registrierungsseite dazu verwendet wird, zu überprüfen, ob ein Nutzername bereits vergeben ist. Dies geschieht durch die Auswertung des jeweiligen HTTP-Status-Codes der Antwort (200 für vorhanden, 404 für nicht vorhanden).

Für eine PUT/POST Anfrage an eine REST API existieren folgende JSON-Objekte und ihre Voraussetzungen. Die Antworten auf GET Anfragen enthalten für gewöhnlich dieselben Felder.

**Hinweise** Wenn ein Parameter nicht angegeben wird oder einer angegeben wird, der nicht geändert werden darf(ID, owner etc.), wird dieser bei PUT-Anfragen nicht verändert. Timestamps werden in Millisekunden seit der UNIX-Epoche angegeben.

- User
  - `username` (String): Nur bei POST-Anfragen benötigt (bei PUT im Pfad), darf nicht leer sein.
  - `email` (String): Optional.
  - `password` (String): Pflicht. Dies muss eine 64 Zeichen lange, in Kleinbuchstaben übertragene, hexadezimale Abbildung des SHA256–Hashes des gewünschten Passworts sein.

  Bitte beachten Sie, dass bei GET Anfragen aus Sicherheitsgründen kein Passwort und keine ID zurückgegeben werden.

- Items

  Bei einer GET-Anfrage auf alle Items eines Benutzers werden sowohl alle selbst erstellten, als auch alle ihm zugewiesenen Items zurückgegeben.

  - `id` (Integer): Serverseitig bestimmt, wird nur bei GET-Anfragen zurückgegeben.
  - `title` (String): Pflicht. Der Titel der todo-Aufgabe.
  - `description` (String): Optional. Die Beschreibung der Aufgabe.
  - `done` (Boolean): Optional. Indikator, ob die Aufgabe erledigt ist.
  - `finished` (Timestamp): Serverseitig bestimmt, wird nur bei GET-Anfragen zurückgegeben. Zeitpunkt, an dem der Eintrag über `done` als erledigt markiert wurde.
  - `created` (Timestamp): Serverseitig bestimmt, wird nur bei GET-Anfragen zurückgegeben.
  - `tags` (List(Tag)): Optional. Liste von IDs der zugewiesenen Tags. Wird absolut (nicht relativ) bearbeitet.
  - `owner` (String): Wird nur bei GET-Anfragen zurückgegeben, wird bei POST/PUT-Anfragen über den Pfad bestimmt.
  - `assignee` (String): Optional. Username des Accounts, der dieses Item zugewiesen wurde. Zum Zurücksetzen den eigenen Benutzernamen eintragen.

  Bei GET-Anfragen kann mithilfe folgender Query-Parameter die Suche serverseitig weiter eingeschränkt werden. Bitte beachten Sie, dass nur Items angezeigt werden, für die Sie die entsprechenden Berechtigungen besitzen.

  - `q`: Suchbegriff.
  - `sortOrder`: `asc` oder `desc`.
  - `sortCriteria`: Einer der folgenden Werte: `id, title, description, created`.
  - `tag`: Die ID eines Tags.
  - `time`: Einer der folgenden Werte: `day, current, week, month, year, all`.
  - `owner`: Benutzername des Besitzers.
  - `assignee`: Benutzername der Person, der das Item zugewiesen wurde.

- Tags

- `id` (Integer): Serverseitig bestimmt, wird nur bei GET-Anfragen zurückgegeben.
- `name` (String): Pflicht. Der Name des Tags.
- `color` (String): Pflicht. Der sechstellige hexadezimale Wert der Farbe.
- `items` (List(Item)): Serverseitig bestimmt, wird nur bei GET-Anfragen zurückgegeben.
- `owner` (String): Wird nur bei GET-Anfragen zurückgegeben, wird bei POST/PUT-Anfragen über den Pfad bestimmt.

# 6 Sicherheitskonzept

Unsere Webapplikation basiert, den Vorgaben entsprechend, auf Apache Shiro. Es existieren zwei Shiro Realms, ein Realm für die "normale" Benutzerauthentifizierung und -autorisierung, und ein Realm für den Administrator-Account. Während das normale *TudoRealm* seine Daten (Benutzeraccounts, Berechtigungen, etc.) in der Datenbank (siehe Kapitel Datenverwaltung) speichert, liest das *TudoAdminRealm* lediglich eine Datei mit dem SHA256-Hash des Administrator-Passwortes ein. Dies hat den Vorteil, dass dieser Account nicht "verloren" ist, wenn die Datenbank beschädigt oder gelöscht wird. Die Unterscheidung zwischen den Realms hat den Implementierungsprozess dahingehend vereinfacht, dass man die notwendige Logik für die Identifizierung des Administrator Accounts in eine eigene, relativ kleine, Klasse auslagern konnte und der normale Realm nicht zu unübersichtlich wurde. Das TudoAdminRealm überprüft so lediglich das Passwort und gewährt alle Berechtigungen. Hierbei ist bei der Entwicklung das Problem aufgetreten, dass das *TudoAdminRealm* allen Benutzern Administratorrechte gewährt hat, bei der Implementierung musst also darauf geachtet werden, dass bei der Methode für die Autorisierung erneut auf den korrekten Benutzernamen "admin" geprüft wird.

Das Realm für die normalen Benutzer beginnt mit der Authentifizierung dahingehend, dass es zuerst prüft, ob Benutzername und Password nicht-leer sind und die richtige Länge haben (64 Zeichen, da dies ein mit SHA256 gehashter String ist). Ist dies geschehen wird der entsprechende Benutzername per UserDAO aus der Datenbank geladen, mit ihm das entsprechende gehashte und gesaltete Passwort. Diese Informationen werden dann als *SimpleAuthentificationInfo* zurück an Shiro gegeben. Als CredentialsMatcher wird der Shiro-eigene *PasswordMatcher* genutzt, dies hat den Vorteil, dass wir uns nicht mehr selbst um das hashen und salten der Passwörter kümmern müssen, was zu etwaigen Sicherheitslücken führen könnte.

Für den *TudoAdminRealm* wird als CredentialsMatcher der *Sha256CredentialsMatcher* genutzt. Dies hat mehrere Gründe, einerseits ist ohne Datenbank ein Saltingprozess nicht wirklich notwendig, andererseits kann man so einfacher im Falle eines Verlustes das Administratorpasswort ändern (man muss nicht durch Shiro neue PasswordMatcher Anmeldedaten generieren, sondern kann ein einfaches SHA256-Skript benutzen). Bei der Administrator-Anmeldung ist auch zu beachten, dass das Passwort ungehasht übertragen werden muss, da der *Sha256CredentialsMatcher* dies erledigt. Wir haben uns für diesen Weg entschieden, da so einerseits der Entwiclungsprozess vereinfacht wurde (man musste nur noch das Administrator-Passwort im REST-Client angeben) und andererseits die Anmeldung im Browser auf unserer Webapplikation nicht mehr möglich ist. Diese würde zu Fehlern führen, da der Administrator-Account bei REST-Anfragen andere Werte zurück liefert als normale Accounts und unser Angularprojekt nicht dafür eingerichtet ist.

Einer der Grundsätze bei der Erstellung unserer Webapp war, dass ein Klartextpassword niemals den Computer (bzw. Browser) des Clients verlassen sollte. Deshalb wird, bevor das eigentliche Anmelde/Registrierformular abgeschickt wird, der Passwort String noch im Browser per Javascript (über die Stanford JavaScript Cryptography Library) mit SHA256 gehasht. Dadurch wird es Angreifern erschwert, das Passwort in einem Netzwerkmitschnitt der HTTP-Anfragen mitzulesen. Selbst wenn dies mitgelesen wird, ermöglicht es höchstens die Anmeldung bei unserer Webapplikation, wenn der Nutzer allerdings auf verschiedenen Websites das selbe Passwort benutzt, und der Angreifer das Klartextpasswort mitgelesen hätte, könnte er sich

auch dort anmelden. Dies wird durch unseren Ansatz vermiden. Ein weiterer Weg, den Übertragungsprozess sicherer zu gestalten bestand darin, HTTPS zu benutzen, so könnte höchstens das Password ausgelesen werden, wenn der Computer des Nutzers bereits kompromittiert wurde. Die Konfiguration von HTTPS war unserer Meinung nach allerdings außerhalb des Rahmens für dieses Projekt, außerdem müsste dies auf den jeweiligen Hosts und nicht an unserem Projekt konfiguriert werden.

Die Berechtigungen werden nicht in der Datenbank gespeichert, sondern zur Laufzeit neu erstellt. Dies ist möglich, da die Berechtigungen bei dieser Art von Projekt noch sehr einfach und leicht generierbar sind: Jeder Nutzer hat die Berechtigungen `item:*:<username>` und `tag:*:<username>`. Des Weiteren erhält jeder Benutzer noch Berechtigungen für die ihm zugewiesenen Einträge. Dies geschieht über eine Datenbankanfrage nach Items, die als "assignee" den jeweiligen Benutzernamen haben. Die jeweiligen IDs werden dann zu den Berechtigungen hinzugefügt.

Auf Rollen wird in unserem Projekt weitgend verzichtet, da wir sie für nicht wirklich sinnvoll im Sachzusammenhang befunden haben. Die einzig existierende Rolle ist die "admin" Rolle, die Zugriff auf einige versteckte REST-APIs bietet. Die einzige Begründung verschiedene Rollen zu implementieren bestünde darin, wenn es verschiedene Benutzer-"Level" geben würde, zum Beispiel "Free User", "Basic User" und "Pro User", die Website also verschiedene Preisstufen und Abonnements anbieten würde. Dies geht allerdings unserer Meinung nach über den Rahmen dieses Projektes hinaus.

# 7 Datenverwaltung

Die Datenverwaltung ist in unserer Webapplikation relativ einfach gehalten. So existieren drei Klassen, `User`, `Item` und `Tag` in `de.webtech.tudo.model`, die alle per Java Annotationen als Hibernate JPA Klassen gekennzeichnet sind. Die Tabellenbezeichnungen sind jeweils der Plural des Klassennamens. Hibernate JPA ist so konfiguriert, dass es automatisch eine H2 Datenbank im jeweiligen Benutzerorder (`C:/User/<username>` bzw. `/home/<username>`) des Benutzers, der den Server ausführt. Die drei Klassen beinhalten neben einer jeweiligen generierten ID als Primary Key und anderen "normalen" Attributen auch Beziehungen zwischeneinander. So existiert eine OneToMany-Beziehung zwischen User und Items, eine OneToMany-Beziehung zwischen User und Tags, und eine ManyToMany-Beziehung zwischen Items und Users. An dieser Stelle ging im Entwicklungsprozess sehr viel Zeit verloren, da wir viele Konfigurationen für die jeweiligen Beziehungen ausprobieren mussten. Besonders zeitaufwändig war hierbei die ManyToMany-Beziehungen, da wir bei der Recherche online auf sehr viele verschiedene Lösungsansätze gestoßen sind, sich alle allerdings in kleinen Details unterschieden haben, sodass wir anfangs keinen Ansatz weiter verfolgen konnten. So scheiterten die meisten Ansätze daran, was passiert, wenn man ein Item löscht: Bei manchen passierte überhaupt nichts, bei manchen wurden alle assoziierten Tags mit gelöscht, und in einem Fall sogar die gesamte Datenbank. Nachdem wir einige Tage an diesem Problem fest saßen, und nach vielen gelesenen Dokumentationen und Foren-Beiträgen, auf einen guten Lösungsansatz gestoßen sind, funktionierte alles.

Die gefundene Lösung für das ManyToMany-Problem besteht aus folgender Konfiguration in der `Item.java`-Datei:

```
@Column
@ManyToMany(fetch = FetchType.EAGER,
        cascade = {CascadeType.MERGE})
@JoinTable(name="tags_items",
        joinColumns = {@JoinColumn(name="item_id")},
        inverseJoinColumns = {@JoinColumn(name="tag_id")})
@JsonIgnoreProperties("items")
private Set<Tag> tags = new HashSet<>();
```

Und folgender Konfiguration auf der `Tag.java`-Seite:

```
@Column
@ManyToMany(fetch = FetchType.EAGER,
        cascade = {CascadeType.MERGE},
        mappedBy = "tags")
private Set<Item> items = new HashSet<>();
```

Dieser Lösungsansatz erstellt eine neue Tabelle namens `tags_items`, welche bei der Suche nach Items automatisch die Item-Tabelle mit der eben beschriebenen Tabelle joint. Diese neue Tabelle enthält nur zwei Spalten, einerseits die Item Foreign Keys und andererseits die Tag Foreign Keys.

Um auf diese Daten auch zugreifen zu können, haben wir, wie im Kapitel "Projekt- und Java-Aufbau" geschildert, entsprechende DAO-Klassen erstellt. Diese erben alle von der abstrakten `GenericDAO`-Klasse, welche Standardfunktionen, wie zum Beispiel `add(Object o)` oder `delete(Object o)` enthält. Die einzelnen Klassen enthalten weiterhin spezifische Funktionen, wie "Alle Tags eines Users" laden. Die `ItemDAO`-Klasse enthält weiterhin Referenzen zur `ItemRequestBuilder`-Klasse, um erweiterte Funktionalität zu liefern.

# 8 Seitenaufbau

Für den Seitenaufbau hat unsere Gruppe einen etwas anderen Ansatz gewählt, als zum Beispiel durch die Beispielprojekte gegeben war. So ist das komplette Angular-Frontend nur erreichbar, wenn der Benutzer schon angemeldet ist, heißt die Anmeldung wird auch nicht von Angular gemanaget. Ist der Benutzer angemeldet, so erhält er entsprechende Cookies (die Apache Shiro `JSESSIONID`, und den eigenen Benutzernamen im Cookie `username`), die er bei allen folgenden AJAX Anfragen vom Frontend aus mit an den Server sendet.

Für die Anmeldung (und Registrierung) stehen zwei einfache HTML Seiten (die durch Java Server Pages generiert werden) zur Verfügung. Die Funktionsweise dieser beiden Seiten ist im Abschnitt "An- und Abmeldeprozess" erläutert.

# 9 An- und Abmeldeprozess

Im Folgenden werden wir den Anmeldeprozess schrittweise ausführlich erläutern.

Der Prozess beginnt damit, dass der Benutzer selbst, oder durch einen Link von einer externen Seite, wie zum Beispiel einer Suchmaschine, unsere Webapp unter `http://<server>/tudo` ansteuert. Da der Nutzer in diesem Beispiel Szenario unsere Seite noch nie Besuch hat, ist er noch unauthentifiziert. Den Server erreicht diese GET Anfrage, und er bereitet sich darauf vor, eine in der `/WEB-INF/web.xml` definierte welcome-file zurückzugeben. Da der Server diese Liste von oben nach unten abarbeitet und überprüft, welche Dateien vorhanden sind, trifft er schon beim ersten gefundenen Eintrag, der `/index.jsp` auf Erfolg. Diese Seite kann dem Benutzer allerdings nicht ausgegeben werden, da die Shiro Konfiguration `shiro.ini` dies nicht zulässt. Stattdessen leitet diese den Server dazu an, die Seite `/account/login.jsp` zurückzuliefern. Diese JSP-Datei überprüft zuerst, ob der Nutzer schon authentifiziert ist (es könnte ja auch ein wiederkehrender, bereits angemeldeter Besucher sein). In diesem Fall würde der Nutzer per HTTP Redirect zu dem Standard Webapp-Pfad `/tudo/` weiterleiten, von dort aus würde der Server ihm, über den eben beschriebenen welcome-file-Prozess, das Angular Projekt zurückliefern.

Da in diesem Beispiel der Nutzer aber nicht authentifiziert ist, erscheint für ihn nur das Anmeldeformular. Sobald der Nutzer dieses ausgefüllt hat, und auf "Sign In" klickt, wird die Formularübertragung allerdings von einem JavaScript-Event-Handler unterbrochen. Dieser sorgt dafür, dass das Benutzerpasswort über die Stanford JavaScript Cryptography Library verschlüsselt wird und in ein verstecktes Input-Feld mit dem Namen "password" eingetragen wird. Des Weiteren wird im DOM der Inhalt des Feldes mit dem ungehashten

Passwort entfernt, damit dieses nicht auch mitgesendet wird. Dies würde zwar keinen Einfluss auf die Anmeldung haben, birgt allerdings Sicherheitsrisiken (weiteres siehe Abschnitt "Sicherheitskonzept"). Sobald dies alles geschehen ist, schickt JavaScript das veränderte Formular per POST-Anfrage an die "login.jsp" Seite.

Von nun an überprüft der Server hintereinander die einzelnen Shiro-Realms, ob die Anmeldung erfolgreich war. Das erste Realm ist das *TudoAdminRealm*, welches lediglich überprüft, ob der angegebene Benutzername "admin" ist. Aus den im Kapitel "Sicherheitskonzept" beschriebenen Gründen, ist die Adminsitrator-Anmeldung vom Webinterface aus nicht möglich. Da somit die Authentifizierung im *TudoAdminRealm* fehlschlägt, wird das nächste Realm, das normale *TudoRealm* angefragt. Dieses Realm führt grundsätzliche Checks aus, wie zum Beispiel die Überprüfung nach fehlenden oder leeren Feldern. Wenn dieser Fall eintritt, wird eine Exception geworfen und der Nutzer wird so behandelt, als hätte er falsche Anmeldedaten angegeben (folgt). Ist die Anfrage korrekt formatiert, überprüft Shiro über die `de.webtech2.tudo.dao.UserDAO`, ob der Benutzername in der Datenbank existiert, und wenn ja, lädt das entsprechende gehashte und gesaltete Passwort. Dieser Benutzername und der Passworthash/Salt werden dann mit den vom Benutzer angegeben Daten abgeglichen. Die Verwaltung der Hashes und Salts wird von dem `org.apache.shiro.authc.credential.PasswordMatcher` geregelt. Dieser kümmert sich auch um die Salt-generierung bei der Registrierung.

Stimmen Benutzername und Passwort, so wird ein `JSESSIONID`-Cookie gesetzt, und der Nutzer an die URL `index.jsp` weitergeleitet. Von hier aus wird der Nutzer an `web/index.jsp` weiter geleitet. Diese Java Server Page ist lediglich dazu da, den Benutzernamen in einem Cookie zu speichern, damit dieser von unserer Angular Applikation für weitere REST-Anfragen genutzt werden kann. Ist dies geschehen, bindet diese `index.jsp` nach dem kurzen Java-Snippet die Angular index.html Datei ein.

Stimmen allerdings Benutzername und Passwort nicht, oder ist es zu einem anderen Fehler bei der Anfrage gekommen, so wird wieder die Login-Seite zurückgegeben. Diese überprüft über ein JSP-Snippet, ob ein fehlerhafter Anmeldeversuch vorlag, und zeigt gegebenenfalls eine Fehlermeldung über die falschen Anmeldedaten an. Die Angular Applikation kann weitere Elemente (wie zum Beispiel die einzelnen Todo-Items) per AJAX-Anfrage nachladen. Die Anfrage-URL setzt sich aus der normalen REST-URL und dem im Cookie gespeicherten Benutzernamen zusammen. Damit diese Anfragen auch authentifiziert sind (die REST-Endpunkte erfordern eigentlich eine BasicHTTPAuthentification), sendet die AJAX-Anfrage die Shiro `JSESSIONID` aus dem bei der Anmeldung gesetzten Cookie mit. Der Ablauf geschieht analog bei POST- und anderen Anfragen. Ist der Besucher fertig mit der Benutzung der Webapp, klickt er auf den "Logout"-Link, welcher ihn zur Seite `/account/logout.jsp` weiterleitet. Diese existiert zwar nicht, ist aber von Apache Shiro als Logout-Seite registriert, sorgt also dafür, dass die Shiro Session aufgelöst wird. Ist dies geschehen, wird der Benutzer auf die Anmeldeseite weitergeleitet und der Prozess beginnt erneut. Der "username"-Cookie kann an dieser Stelle ignoriert, da nur die Angular Applikation ihn nutzt. Selbst wenn der Nutzer sich mit einem anderen Benutzernamen anmeldet, wird der Cookie einfach überschrieben.

# 10 Registrierungsprozess

Der Registrierungsprozess ist im Vergleich zum Anmeldeprozess relativ einfach gehalten. Beim Aufruf der entsprechenden Seite wird dem Benutzer ein Formular im gleichen Stil wie das von der Anmeldeseite zurückgegeben. Alle Felder haben JavaScript Events definiert, so dass beim Tippen schon auf etwaige Fehler überprüft wird, wie zum Beispiel, ob die beiden Passwörter übereinstimmen. Es wird außerdem über eine AJAX-Anfrage geprüft, ob der eingegebene Benutzername bereits vorhanden ist. Ist dies der Fall, wird dem Benutzer eine Fehlermeldung angezeigt, die ihn darauf aufmerksam macht und verhindert, das Formular abzusenden.

Stimmen alle eingegebenen Daten und versucht der Nutzer das Formular abzusenden, wird dies von einem weiteren JavaScript Event aufgehalten. Dieses führt nun, wie im Anmeldeprozess dazu, dass das Passwort gehasht wird, und die entsprechenden Daten per AJAX-POST-Anfrage an den Server gesendet wer-

den. Liefert der Server den HTTP-Status 201 CREATED zurück, so wird dem Nutzer eine Erfolgsmeldung angezeigt,andernfalls wird ihm eine Fehlermeldung angezeigt.

# 11 Entwicklungsprozess

## Allgemein

Beim Entwicklungsprozess haben wir eine vielzahl an Tools und IDEs ausprobiert, um uns die Arbeit so angenehm wie möglich zu machen. Nennenswert dabei sind Apache Maven, welches das Deployment sehr erleichtert hat, und IntelliJ IDEA Ultimate, welches uns im Gegensatz zu anderen IDEs nicht im Weg stand und uns die Arbeit sehr vereinfacht hat, nicht zuletzt dadurch, dass der Entwickler JetBrains eine kostenlose Ultimate-Version für Studenten anbietet.

## Backend

Für die Entwicklung des Backends haben wir, wie bereits beschrieben, das Projekt ohne das Maven Dart Plugin gebaut, sodass Anmeldung, Datenverwaltung und REST API testbar waren, ohne lange Kompilierungszeiten in Anspruch nehmen zu müssen. Zum testen der REST API kam *Insomnia REST Client* zum Einsatz, welches wir im Vergleich zu den Alternativen, wie zum Beispiel Postman, als sehr angenehm und einfach empfunden haben.

## Frontend

Um bei kleinen Änderungen am Frontend (zum Beispiel eine minimal veränderte `.css`-Datei, wie es sehr häufig bei der Webentwicklung vorkommt) testen zu können, ohne jedes Mal erneut das komplette Maven Projekt bauen zu müssen, kam die *serve*-Funktionalitat von NodeJS bzw. Dart zum Einsatz, so konnten wir über den Befehl `pub serve` sehr einfach Änderungen vollziehen, die innerhalb von Sekunden im jeweiligen Browser dargestellt wurden.

# 12 Weitere Probleme

**Instanziierung des *EntityManagers*** Eins der großen Probleme war die Erreichbarkeit des *EntityManagers*, da dieser nur einer Klasse, die über eine Java Bean erstellt wurde, injiziert werden konnte. Dieses Problem war wahrscheinlich, neben dem "Datenstruktur" Problem, das größte Problem an diesem Projekt, da wir mehrere Klassen brauchten, die dynamisch erstellt werden konnten und trotzdem Zugriff auf den *EntityManager* haben. Nach etwa einer Woche Recherche und Lösungsversuchen, blieb uns keine andere Wahl, als den Code aus den Beispielprojekten zu modifizieren. Unsere Lösung besteht nun aus einer Klasse `DAOCreator`, die über eine Methode Subklassen der `GenericDAO` als Java Bean generieren kann.

**Zeitmanagement** Das nächste Problem, welchens nicht einmal an der Technik lag, war der Zeitdruck. Nachdem wir monatelang fast täglich jeweils meherere Stunden an diesem Projekt saßen, sind wir erst am Dienstag vor der Deadline mit der kompletten Funktionalität fertig geworden. Dabei mussten einige Features aufgegeben werden, zum Beispiel die geplante Datenanalyefunktion (siehe "Erste Gedanken"). Dies könnte auch die Tatsache als Ursache haben, dass wir statt den empfohlenen vier Personen pro Gruppe nur zwei waren. Nachdem wir an diesem Dienstag das Projekt größtenteils fertig gestellt hatten, starteten wir mit der Verfassung des Berichtes und mit der Anwendung der *Automata Learning Experience*. An dem eigentlichen Projekt wurden von nun an nur noch Bugfixes (meistens durch das Testing mit *ALEX* gefunden) durchgeführt und eine `README.md` verfasst.

**Verbesserungsmöglichkeiten** Ein Problem fiel uns erst relativ weit am Ende des Entwicklungsprozesses auf, nämlich das Testen. Bis zu diesem Punkt hatten wir alles von Hand mit einem REST-Client getestet, was sich aber gegen Entwicklungsende mit sämtlichen Testfällen und Konfigurationen als schwierig erwies. Unser Plan bestand nun daraus, JUnit Tests einzuführen, die direkt die Methoden aus der REST-API aufrufen,

dies war allerdings nicht möglich, da der Test *vor* der Ausführung auf dem Server stattfinden würde, er aber den Server braucht, da er sonst keinen Zugriff auf die Datenbank hat. Da uns gegen Projektschluss nicht genug Zeit blieb, um dieses Problem ordentlich zu lösen (es gibt bei solchen weit verbreiteten Technologien wahrscheinlich bessere Lösungsmöglichkeiten), haben wir uns auf REST-Tests mit ALEX beschränkt.

# 13 Automata Learning Experience

## Vorbereitung

Bevor wir uns Gedanken gemacht haben, welche Schnittstelle wir lernen wollten, begannen wir zuerst ein Grundgerüst aufzustellen. Das bedeutet also, wir haben uns überlegt, welche Dinge kann ein User auf unserer Seite machen?

Dabei ergaben sich zuerst folgende Gruppen mit denen man umgehen konnte: Todo Notes, Tags und User. Anschließend ging es darum, diesen Gruppen Symbole zuzuordnen. Wir begannen bei den Usern, da diese den Grundstein für unsere Anwendung legten. Diese konnte man nur erstellen und suchen. Man konnte sie zwar über die Restschnittstelle auch löschen, dies war aber vorerst nur zum Bereinigen der Datenbank für uns beim Testen gedacht. Aufgrund von Zeitgründen haben wir dem Nutzer auf der Browserschnittstelle keine Möglichkeit dazu gegeben. Dies kann natürlich relativ schnell nachgeholt werden.

Dann kam die Gruppe der Tags, die zweitkleinste Gruppe. Tags kann man in unserer Anwendung erstellen, löschen und suchen. Das machte also 3 Symbole.

Zum Schluss füllten wir dann noch die Gruppe der eigentlichen Aufgaben, mit ihnen konnte man einiges mehr machen, zum Beispiel sehr viel mehr einzelne Eigenschaften bearbeiten und die jeweiligen GET-Anfragen erweitert filtern. Bevor wir uns aber an das Füllen des Grundgerüstes machen, mussten wir noch ein Resetsymbol hinzufügen. Dies ist dafür zuständig alle Variablen in den Globalen Kontext zu bringen und hierfür Standardwerte zu setzen.

## Symbole definieren

Zuerst versuchten wir uns an der Browser-Schnittstelle. Dieser Versuch erwies sich allerdings als sehr problematisch und zeitraubend. Die Adressierung mittels CSS-Selektoren klappte nur bedingt, anhand von IDs und Klassen wurden die Elemente leicht ausfindig gemacht, folgte aber ein Input ohne diese Werte, war ein Ansprechen über den CSS-Selektor `#loginForm > input` nicht möglich. Hierbei waren wir uns aber sicher, dass der Selektor der Richtige war, da dieser in unseres CSS Datei exakt so genutzt worden war. Nachdem wir mittels exzessiven Nutzen von IDs und Klassen auf ein funktionierendes Registrierungssymbol kamen, mussten wir aber feststellen, dass das Erstellen hierfür sehr lange dauerte. Zudem hatten wir alleine für das Testen der Registrierung schon 11 Aktionen.

Über die REST-Schnittstelle musste man gerade einmal eine POST-Anfrage absenden und die Antwort prüfen. Ab dort an wandelte sich unser Vorhaben also und wir erlernten die REST-Schnittstelle. Leider begannen auch hier wieder Probleme. ALEX führt am Anfang jeder Anfrage JavaScript aus, um Cookies, local- und sessionStorage zu löschen, um eine Unabhängigkeit von den zuvor durchgeführten Tests zu garantieren. Dies führte bei uns zu einer Fehlermeldung, da wir nur REST Anfragen benutzten und den Browser überhaupt nicht öffnen wollten. Durch ein zusätzlichen Resetsymbol, welches nur den Browser öffnet und wartet bis die Seite geladen ist, verschwand dann auch unsere Fehlermeldung und wir konnten beginnen die Symbole zu füllen. Wie wir uns gedacht haben sind hierfür weniger Aktionen erforderlich, so hatten wir pro Symbol nur noch etwa zwei Aktionen, die jeweils gleich aufgebaut waren: Zuerst erstellt man den notwendigen Request und überprüft anschließend den Status der Antwort.

Anfangs haben wir uns mit den Inputs sehr zurückgehalten und haben nur Username und Passwort genutzt. Doch schnell wurde uns klar, das war viel zu wenig. Wir brauchten die IDs der erstellten Tags und Items, um auch genau diese wieder löschen zu können. Wir wollten zudem auch einen Counter haben, der einfach an den Usernamen angehängt werden kann, um leichter neue Nutzer zu erstellen. Auch einfache Dinge, wie Titel und Status der Aufgabe (fertig oder nicht fertig), wollten wir nicht immer wieder innerhalb des Symbols ändern. Zudem hatte man während dem Testen beim Input die Wahl, ob man die Variable aus dem globalen Kontext lädt oder diese selbst eingibt.

**Hinweis** Im Anhang befindet sich eine Liste von allen Symbolen, die wir definiert haben.

## Testing

Nach dem Erstellen eines Symbols testeten wir es direkt, dazu legten wir bei Test einen Case an der keine weitere Rolle spielen sollte. Wir fügten dort die Resetsymbole (Variablen und Browser) und jeweils ein einziges weiteres Symbol ein. Durch die vielen Inputs konnten wir nun ohne auf den globalen Kontext angewiesen zu sein einzelne Symbole testen.

Um die vollständige Antwort kontrollieren zu können und IDs bereits bestehender Tags oder Items herauszufinden nutzen wir den shcon erwähnten Insomia REST Client. Da man in ALEX den Status zwar überprüfen kann, aber man im Falle eines falschen Status nicht die Möglichkeit hat schnell zu sehen, welcher Status stattdessen gesendet wurde, nutzen wir auch hier während dem Testen zusätzlich Isomnia, dort hatten wir die Möglichkeit einfache Anfrage abzuschicken und uns die Antwort im Detail anzuschauen. Nachdem wir uns sicher sein konnten, dass alle Symbole alleine funktionierten, konnten wir uns relevante Testfälle überlegen.

So testeten wir zum Beispiel, ob man ein Item erstellen kann und es nach dem Löschen auch wirklich nicht mehr zu finden war. Dazu war sehr hilfreich, dass man im Tool angeben konnte, dass ein Symbol scheitern sollte. So konnte man festlegen, dass der Testfall erfolgreich war, obwohl das letzte Symbol, welches ein nicht existierendes Item sucht, nicht erfolgreich war. Außerdem testen wir, ob man ein Tag anlegen kann und es im Anschluss auch im Item verwenden kann. Nach ein paar Tests fiel uns auf, dass unsere Seite das zu tun scheint was sie soll. Doch uns interessierte auch, was passiert, wenn ein User versuchte etwas zu tun, was er nicht durfte. So sollte es natürlich nicht möglich sein, das ein User das Item eines anderen Users bearbeiten oder erstellen kann (Zuweisungen der Items hier ausgenommen).

Also mussten wir alle Symbole erneut überarbeiten. Wir fügten neben einer Variablen für einen Usernamen, eine für einen Loginnamen hinzu. Bei dem Standardwert dieser Variable entschieden wir uns, sie auf denselben Wert wie den Usernamen zu setzen. Da so ohne weitere Angaben immer der richtige Fall ausgeführt wird. Die Authentifizierung fand also nun mit dem Loginnamen und dem Passwort statt, während der Username oben in der URL seinen Platz fand. Durch erneutes Testen, diesmal mit einem anderen Usernamen als Loginnamen, fiel uns zu unserem Unglück auf, dass man die Items anderer Nutzer sehen konnte. Dies mussten wir also zuerst beheben, bevor wir weiter testen konnten (siehe dazu Kapitel “Sicherheitskonzept”). Nachdem unsere Tests erfolgreich waren und wir uns relativ sicher sein konnten, dass unsere Webseite sich so verhielt wie sie es sollte, begannen wir uns mit dem Learning auseinanderzusetzen.

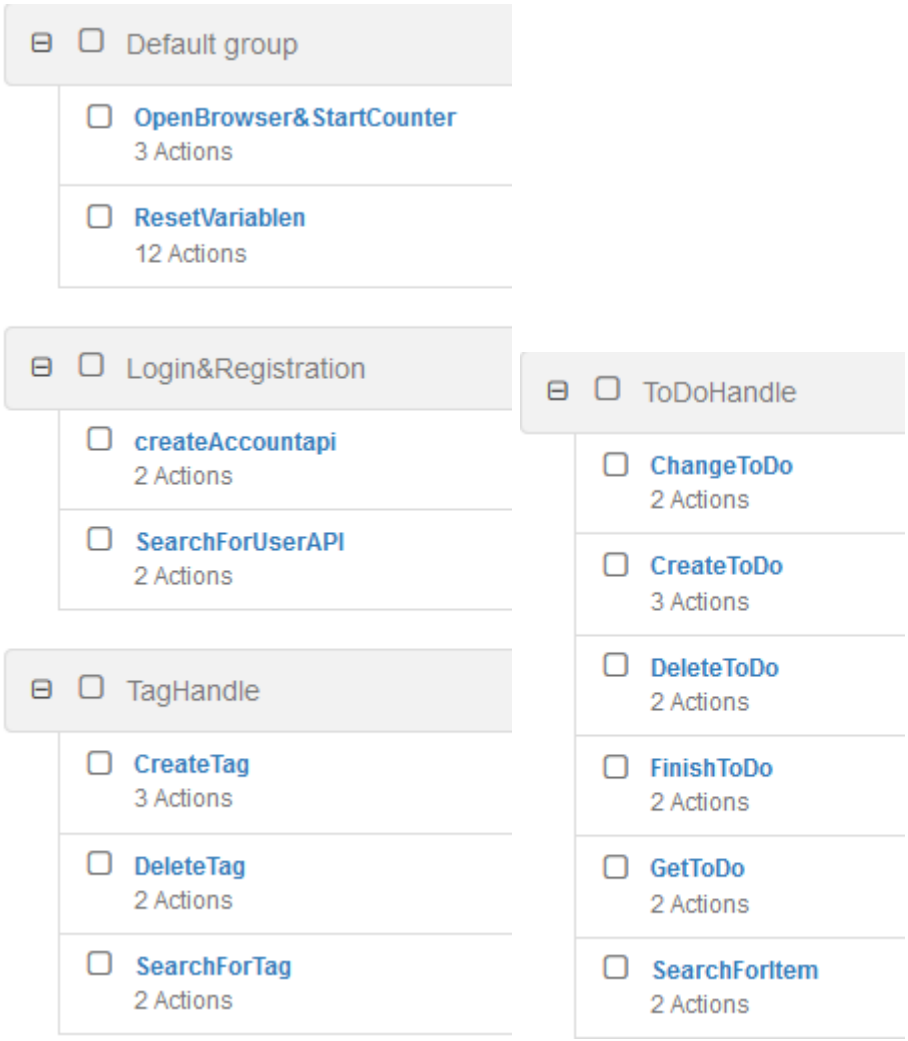
## Learning

Zu unserer Überraschung bekamen wir hier keine Fehlermeldungen bezüglich Sicherheitsprobleme, obwohl wir unser Symbol für das Öffnen des Browsers hier nicht als Resetsymbol nutzen konnten, da hier schon unser Reset für die Variablen eingesetzt wurde. Wir begannen also mit den Standardeinstellungen unseren ersten Lernprozess, hierfür nutzten wir alle Symbole der Gruppen: User, Tag und Todo Notes, außer das GetTodos, da dies alle Todo Notes ausgibt und immer gelingt, sobald der Nutzer existiert. Dies ist mit einem Symbol SearchForItem sinnvoller, da dieses nach einer bestimmten ID sucht.

Nach ein paar Versuchen bekamen wir endlich einen Automaten mit mehr als zwei Zuständen heraus, welcher natürlich absolut fehlerhaft war. Wir hatten nun einen Automaten mit 5 Zuständen (siehe Anhang). Die Funktionalität unserer Webseite hatte er gut eingefangen. Aber schnell stellten wir fest, dass es hierbei nur schwer möglich war zu kontrollieren was passiert, wenn man fehlerhafte oder boshafte Anfragen sendet. Da wir dies aber in der Testphase erledigt hatten, war das kein Problem. So zeigte der Automat an, dass man zuerst Dinge erstellen musste bevor man sie löschen oder ausgeben kann oder das man für jegliche Aktionen eingeloggt bzw. registriert sein muss. Was man allerdings nicht sieht ist, wie der Umgang mit alten Daten funktioniert. So ist es natürlich möglich durch eine HEAD- Anfrage zu überprüfen, ob ein Nutzer schon existiert, HEAD wird allerdings leider nicht von ALEX unterstützt. Eine weitere Beschränkung bestand daraus, dass es zwar möglich ist zwei Todo Notes zu erstellen und beide wieder zu löschen. Der Automat hat allerdings nur die ID des zuletzt erstellen Todos und daher kann er das zweite nicht löschen. Generell kann man aber sagen, dass der Automat die Situation eines neu registrierten Nutzers gut darstellen kann. Vielleicht hätte sich das Problem mit der Handhabung der alten Daten noch lösen können, dazu blieb uns aber nicht genug Zeit.

# 14 Anhang

## *ALEX* - Symbole



## *ALEX* - Automat