Projektabschlussbericht

Platform zur Erstellung, Bearbeitung und Kommentierung von Internet Standards

Erstellung eines Online-Werkzeugs für gemeinsames Schreiben von Dokumenten. Internet Standards werden von der IETF, der Internet Engineering Task Force, geschrieben. Die Werkzeuge um diese Dokumente zu erstellen sind umständlich zu bedienen und gestalten gemeinsames Schreiben von Dokumenten schwierig. Das muss besser gehen!

Daher ist Ziel dieses Projektes der IETF zu helfen, indem ein Online-Werkzeug erstellt werden soll, dass gemeinsames Schreiben von Dokumenten nach dem WYSIWYG (What you see is what you get) Prinzip ermöglicht. Zusätzlich soll es möglich sein existierende Dokumente zu kommentieren und zu durchsuchen. Ein Account-Verwaltungssystem soll erstellt werden, damit die Bearbeitung von Dokumenten nur autorisierten Personen gestattet ist. Eine Datenbank und eine Weboberfläche sollen gestaltet werden, um Dokumentabhängigkeiten zu visualisieren. Verschiedene Werkzeuge sollen regelmäßig und automatisiert die Datenbestände mit der IETF abgleichen.

Projektbetreuer:
Prof. Dr. Rolf Winter

Projekt-Team:

Sanim Rashid	Hannah Walkiw	André Majeres
Sebastian Pröll	Denis Richter	Rafael Schmitt
Max Schleuniger	Michael Jünger	

Inhaltsverzeichnis

1 *Projektziel:*

Internet Standards werden von der IETF, der Internet Engineering Task Force, geschrieben. Ziel unserer Projektarbeit war, ein Online-Werkzeug zu erstellen, dass gemeinsame Schreiben von Drafts ermöglicht. Des Weiteren soll es die Möglichkeit geben, existierende RFCs zu kommentieren und zu durchsuchen. Ein Account-Verwaltungssystem soll erstellt werden, damit die Bearbeitung von Dokumenten nur autorisierten Personen gestattet ist. Das gruppeninterne Ziel sollte stets effektiv, zielorientiert, motiviert und kommunikativ verfolgt werden.

2 Projektteilnehmer

Projektleiter: Professor Dr. Rolf Winter

Projektteilnehmer

Michael Jünger

André Majeres

Sanim Rashid

Hannah Walkiw

Sebastian Pröll

Max Schleuniger

Rafael Schmitt

Denis Richter

3 *Vorgehen und Phasen der Projektarbeit*

Um dieses Ziel in die Tat umzusetzen, haben wir unser Projekt in verschiedenen Phasen unterteilt:

Informationsbeschaffung:

- Über Professor und Internet

Organisation

- Raumbeschaffung, Zeitplanung, Termine

Grobplanung

- Besprechung der des Vorgehens, Analyse der Problemstellung

Aufgabenbereiche

- Arbeitsumgebung eingerichtet
- Aufteilung des Projekts in kleinere Bereiche
- Zuteilung von Aufgaben an die Projektteilnehmer

Testen und Dokumentieren

- Testen der Programmeinheiten und Dokumentation
- Datenbankentwurf

Zusammensetzung

- Protokollierung und Verknüpfung der Programmeinheiten

4 *Aufgabenbereiche und Zuteilung*

- **Einrichten des Servers:**
  Michael Jünger

- **Bearbeitung von Drafts mit Hilfe von Etherpad Lite:**
  Michael Jünger
  Sanim Rashid
  André Majeres
  Denis Richter

- **Kommentieren der RFCs und Errata:**
  Hannah Walkiw
  Sebastian Pröll

- **Durchsuchen der RFCs und Login-System:**
  Rafael Schmitt
  Max Schleuniger

5 *Einrichten des Servers*

Für unser Projekt haben wir einen Server von der Hochschule gestellt bekommen, auf dem bereits ein Betriebssystem (Debian squeeze) vorinstalliert und der Apache-Webserver eingerichtet waren.

Zur weiteren Verwendung des Servers bekamen wir einen root-Zugang per ssh. Dies beinhaltete sowohl die Pflicht der Instandhaltung (regelmäßige Updates, Beachung der Sicherheitsaspekte der verwendeten Software und einrichten der von uns benötigten Komponenten), sowie die Freiheit den Server nach unseren Bedürfnissen einrichten zu können.

Damit jeder Projektteilnehmer auch Zugang zum Server hatte, mussten zunächst Benutzer angelegt, sowie die Clients der Teilnehmer eingerichtet werden. Dies beinhaltete die Konfiguration des ssh-Clients, das Anlegen eines public-keys (client-/serverseitig), Einrichten eines git-Repositories (serverseitig) mit Zugang mittels gitosis und dem public-key, Installieren einer Weboberfläche für git, Einrichten der mysql-Datenbank inklusive Weboberfläche zur Administration und der Installation der benötigten Frameworks (Django und Nodejs).

Weiter benötigten wir noch die aktuellen Drafts von der IETF, welche wir mittels rsync-Protokoll und zuhilfenahme des cron-Dienstes aktualisierten. Mithilfe des cron-Dienstes realisierten wir auch regelmäßige Backups.

Im Verlauf der Projektarbeit mussten einzelne Dienste neu gestartet werden, da Änderungen am Quellcode sonst nicht wirksam geworden wären. Zur Vereinfachung wurden dazu verschiedene Shell-Skripte geschrieben.

Bei der Einrichtung des Servers gab es keine großen Schwirigkeiten, jedoch aber bei dessen Verwendung durch die verschiedenen Benutzer. Die Bearbeitung der Daten durch verschiedene Benutzer hatte zur Folge, dass nicht alle Benutzer zu jeder Zeit Schreibrechte auf alle Dateien hatten. Legte zum Beispiel Benutzer A eine neue Datei an, konnte Benutzer B diese nicht schreiben, da Linux automatisch den Ersteller als Eigentümer einrichtet und nur Eigentümer mit Schreibrechten versieht. Selbst dann, wenn das übergeordnete Verzeichnis Gruppenschreibrechte vorsieht.

Damit nun nicht mehr regelmäßig die Schreibrechte manuell angepasst werden mussten, um allen Benutzern Schreibrechte auf die Quelldateien zu gewähren, hatten wir einen weiteren Benutzer „project“ angelegt, mit dem sich fortan alle Teilnehmer angemeldet haben. So lagen die Schreibrechte aller Dateien stets beim Benutzer „project“ und jeder konnte ungestört arbeiten.

6 *Bearbeitung von Drafts mit Hilfe von Etherpad Lite:*

6.1 *Einrichten der Komponenten und Einarbeitung:*

Unsere Aufgabe bestand darin, mehreren Personen die Möglichkeit zu bieten, gleichzeitig an einem Dokument zu arbeiten. Da Etherpad Lite, ein webbasierter Editor zur kollaborativen Bearbeitung von Texten es mehreren Personen erlaubt, in Echzeit einen Text zu bearbeiten, haben wir uns dafür entschieden, diesen in unser Projekt einzubinden.

Etherpad Lite ist in JavaScript programmiert und basiert auf Nodejs. Zunächst war die Aufgabe Etherpad Lite auf unserem Server zu installieren und sich in den Quellcode von Etherpad Lite, sowie in JavaScript und in das Framework Nodejs einzuarbeiten. Anschließend haben wir unsere eigene Datenbank eingebunden. Hierfür mussten zuerst die benötigten Tabellen in MySQL erstellen und danach Änderungen in der *settings.json* Datei vornehmen.

6.2 Entwicklung eines Datenbankmodules:

Da wir in unserem Projekt auch eine eigene, von Etherpad unabhängige Datenbank verwenden, haben wir zur Vereinfachung ein Modul geschrieben, welches es uns auf einfache Weise und mit Unterstützung des *mysql*-Modules ermöglicht, Anfragen an diese zu stellen und die Ergebnisse zu verarbeiten. In diesem Modul gibt es drei Methoden, *initConData(database, username, password), initStatement(statement) und* dbQuery(callback). Die ersten beiden Methoden initialisieren die Datenbank anhand des übergebenen Benutzernamen und Passwortes, sowie das SQL-Statement und müssen aufgerufen werden bevor die Methode *dbQuery(callback)* aufgerufen wird. Ob dies geschehen ist, wird beim Aufruf der dbQuery-Methode geprüft. In der dbQuery-Methode wird dann die Anfrage über das mysql-Modul an die Datenbank gesendet und das Ergebnis in der Variable *results* gespeichert. Der Parameter *callback* der dbQuery-Methode enthält eine Funktion, die bei erfolgreicher Abrfage der Datenbank ausgeführt wird. Was genau die callback-Methode macht und wie sie die Daten aus der Variable *results* verarbeitet, bleibt dem Anwnder überlassen. Diese Flexibilität war es letztenendes, die uns dazu bewegt hat dieses Modul zu schreiben.

6.3 Entwicklung eines Modules zum Dateiimport:

Ein Problem während der Entwicklung war es, dass wir zum einen bereits geschriebene Drafts in Form von Textdateien hatten, welche wir aber zur Bearbeitung in Etherpad in dessen eigene Datenbank importieren mussten. Das Importieren aller Dateien funktionierte jedoch leider nicht, da das node-Framework selbst nur einen Thread für alle Aufgaben verwendet. Da dies zur Folge hätte, das ein Benutzer alle anderen Benutzer blockieren kann, wenn dieser eine Anfrage an den Server stellt, die länger dauert, verwendet nodejs *asynchronous callbacks*. Das bedeutet, dass eine Aufgabe in den Hintergrund gestellt und später ausgeführt wird. Liegt das Ergebnis vor, so wird per callback angezeigt, dass die Methode ihre Aufgabe erledigt hat. So wäre dies auch bein Importieren der Drafts in die etherpadeigene Datenbank der Fall gewesen. Bei mehr als 70.000 Drafts, die alle in den Hintergrund ausgelagert wurden, gab es bereits bei c.a. 10000 Drafts erhebliche Probleme mit der Speicherauslastung. Deshalb entschieden wir uns die Drafts nicht auf einmal zu importieren, sondern nur bei Bedarf, wenn ein Besucher diesen Draft lesen oder bearbeiten möchte. Das Modul zum Importieren ist gegenüber des vorangegangenen Problemes trivial. Es verwendet lediglich ein Modul *fs (filesystem)* mit dessen Hilfe es die Datei, deren Dateiname per Übergabeparameter übergeben wird einliest, soweit diese vorhanden ist. Ist dies der Fall, gibt die Methode den Text der Datei zurück, existiert die Datei nicht, gibt sie *NULL* zurück. Etherpad selbst entscheidet dann anhand des Rückgabewertes ob der Text der Datei oder der Standardtext (bei Rückgabewert *NULL*) in das neu erstellte Pad geladen wird.

6.4 Modul zum Versenden von E-Mails

Um den Besuchern die Möglichkeit zu geben die bearbeiteten Drafts bei der IETF einzureichen, haben wir ein E-Mail-Modul geschrieben. Es benutzt intern das Modul *nodemailer*, bietet aber die Möglichkeit den Mailserver zu konfigurieren. Eine Methode *sendMail(from, to, subject, text, html)* nimmt die benötigten Daten entgegen und gibt sie an die zuständige Methode von *nodemailer* weiter. Dies hat den Vorteil, dass der Mailserver nur einmal konfiguriert werden muss und E-Mails flexibel von Person A an Person B versendet werden können.

6.5 Request-Parser Modul:

Um Anfragen aller Art bearbeiten zu können, verwendet Etherpad in seiner *server.js*-Datei eine Methode app.get(„URL“, function(req, res) {...}), mit deren Hilfe Anfragen an eine URL, die im ersten Parameter definiert ist, bearbeitet werden können. Der zweite Parameter, beschreibt die Funktion, die letztenendes die gewünschten Aktionen bei einer Anfrage an diese URL ausführt. Da mittels der *GET*-Methode bei einer URL-Anfrage auch sogenannte Schlüssel-Wertepaare mit der URL an den Server gesendet werden können, haben wir zur vereinfachten Verwendung dieser das Request-Parser Modul geschrieben. Dieses besitzt lediglich eine Methode *parseURL(req)*, die bei ihrem Aufruf das *request*-Modul übergeben bekommt. In diesem steckt unter anderem auch die URL des Clients inklusive der Schlüssel-Wertepaare. Eine solche URL ist folgendermaßen aufgebaut: *http://example.com?key1=value1&...&key$_x$=value$_x$*. Die Schlüssel-Wertepaare sind dabei durch ein '?' von der URL abgetrennt, die einzelnen Paare untereinander mit einem '&'. Diese Gegebenheiten sind auch der Ansatzpunkt für den Request-Parser. Dieser trennt zunächst die Schlüssel-Wertepaare von der URL ab und dann die einzelnen Paare vom verbliebenen URL-String. Bei jeder Abtrennung werden Schlüssel und Wert in einem assoziativen Objekt gespeichert und der String um das abgetrennte Paar reduziert. Der Request-Parser gibt dann bei Vollendung seiner Arbeit dem Aufrufer das Objekt zurück. So kann der Aufrufer die Daten auf einfache Weise mit deren Schlüssel ansprechen und zugleich überprüfen ob alle notwendigen Felder vom Client an den Server gesendet wurden. Auf diese Art und Weise können vom Client auch komplexere Anfragen mit vielen Schlüssel-Wertepaaren an den Server gesendet werden, die dieser dann bearbeiten kann.

6.6 *Draft-Search Modul:*

Um den Besuchern das Finden der Drafts zu erleichtern, haben wir ein Draft-Search Modul entwickelt. Dieses Modul erhält vom Client ein Suchpattern. Mit diesem sucht das Modul in der Datenbank nach den passenden vorkommen. Da es bei den Drafts jedoch zum einen Drafts ohne Revision und zum anderen verschiedene Revisionen gibt, müssen diese erst herausgefiltert werden, um die Anzahl der Suchergebnisse zu minimieren. Im ersten Schritt wird das vom Datenbankmodul zurückgelieferte assoziative Array in ein indexbasiertes Array kopiert und die Einträge sortiert. Dies übernimmt die Funktion *copyArray(results)*. *Dann* werden aus dem Fundus der Drafts in der Methode *pushDrafts(draftArray)* zunächst die Drafts herausgefiltert, die keine Revisionen besitzen und in ein seperates Array *searchResult* abgespeichert, sowie aus dem Array mit allen Drafts gelöscht. Daraufhin wird an eine weitere Methode *eliminateRev(draftArray, searchResult)* das Array mit allen Drafts und das Array mit den Drafts, die keine Revision besitzen übergeben. Diese löscht dann alle Vorkommenden niedrigerer Revisionen und fügt nur die Drafts mit höchster Revsion in das Array *searchResult* ein. Da die Array-Elemente aber nicht wirklich gelöscht werden, sondern als Null-Type Objekte im Array verbleiben, müssen diese im letzten Schritt noch von der Methode *deleteNullTypeObjects(searchResult)* in ein neues Array umkopiert werden.

6.7 *Das Modul Authenticate:*

Damit nicht jeder Besucher schreibenden Zugriff auf die Drafts hat, benötigten wir noch eine Art Authentifizierungsmechanismus. Da wir in unserem Projekt bereits eine Loginfunktion hatten, welche in Python und dem Django-Framework entwickelt wurde, bedienten wir uns diesem bestehendem System. Wie es genau funktioniert, wird später im Abschnitt Loginsystem näher erläutert. Das autehtifizierungsmodul bedient sich einer von Django generierten Session-ID, die in einem Cookie auf dem Client gespeichert ist und von uns ausgelesen wird.

Anhand dieser Session-ID ist es nun möglich in der Datenbank nach der selben Session-ID zu suchen. Kommt diese vor, ist der Benutzer authentifiziert, also eingeloggt.

Eine Schwäche dieses Systemes ist es, dass die numerische ID des Benutzers, die neben der Session-ID in der Datenbank abgespeichert wird, verschlüsselt ist und es uns nicht mehr gelungen ist diese zu entschlüsseln. Das hat zur Folge, dass wir gegenwärtig nur prüfen können, ob ein Benutzer eingeloggt ist, nicht aber zu welchen Drafts dieser tatsächlich Zugang hat. In der Datenbank sind zu jedem Benutzer auch die Drafts, an welchen dieser arbeitet, sowie dessen numerische ID gespeichert. Mit Hilfe der numerischen ID wäre es uns möglich den Zugang des Benutzers nur auf seine Drafts zu beschränken. Da wir diese aber nicht entschlüsseln und entsprechend gegenprüfen können, können wir noch nicht verhindern, dass ein Benutzer mit der Benutzer-ID „5“ die Drafts des Benutzers mit ID „6“ bearbeiten kann. Es ist also momentan lediglich der Zugang für nicht registrierte Benutzer beschränkt.

6.8 *Weitere Anpassungen in der Datei „server.js“*

Damit die von uns geschriebenen Module auch funktionieren, mussten diese natürlich noch in den Quellcode von Etherpad aufgenommen werden. Die dafür notwendigen Änderungen beschränken sich allerdings bis auf wenige Ausnahmen auf die Datei *server.js*. Die verschiedenen Anfragen an den Server zur Authentifizierung, zum Versenden einer Mail, zur Suche oder dem Lesen von Drafts, realisierten wir über Anfragen an eine extra für den jeweiligen Zweck vorgesehene URL. Etherpad bietet dafür eine Methode app.get(„URL“, function(req, res) {…}), wie bereits beim Request-Parser-Modul erwähnt. Die zum Parameter URL passende Methode wird demnach aufgerufen und die im zweiten Parameter übergebene Funktion verarbeitet dann die Daten entsprechend dem Zweck.

7 *Comment und Errata System / Profil Seite:*

7.1 *Comment und Errata System:*

Da es für registrierte Nutzer möglich sein sollte Internet Standards unkompliziert zu kommentieren und gegebenenfalls Errata zu verfassen, haben wir das Comment/Errata-Modul entwickelt. Nicht registrierte Besucher der Seite können Kommentare und Errata lediglich lesen, keine eigenen verfassen.

Für das Comment/Errata System haben wir nach dem Django Prinzip im MVC-Pattern (Model View Controller) eine eigene App "blog" entwickelt.

- Der Pythoncode zur Programmierung wurde daher in views, *comment()* und *errata()*, aufgeteilt und liegt in *blog.views.py* (Controller).
- Im *blog.templates*-Ordner haben wir alle für die *blog*-App benötigten html-Dateien, *base.html*, *comment.html* und *errata.html* , abgelegt (View). HTML-Code der für *comment.html* und *errata.html* identisch ist haben wir in eine eigene Datei *base.html* ausgelagert, und von dieser dann jeweils abgeleitet *{{% extends "base.html" %}}*, dadurch also Code-Duplikation vermieden.
- Um Kommentare und Errata verwalten und bei Abfrage richtig zuordnen zu können haben wir zwei Datenbank Tabellen *class Comment(models.Model)* und *class Errata(models.Model)* in *models.py* mit jeweils folgenden Attributen angelegt: *username*, *date*, *text*, *rfc* und *line* (Model)
- Um die Kommentare beziehungsweise Errata zu erfassen haben wir das von Django bereitgestellte Modul *forms* verwendet. In der Datei *forms.py* haben wir hierfür zwei Klassen *class CommentForm(forms.Form)* und *class ErrataForm(forms.Form)* erstellt.

Hat der Benutzer einen Internet Standard zum Lesen geöffnet und klickt auf eine der Kommentar-(grün) oder Errata-(rot) Schaltflächen, die für jede Zeile des Internet Standards verfügbar sind, so wird die entsprechende view *comment()* oder *errata()* aktiv. Dieser werden drei Variablen, *request*, aktuelles RFC und Zeilen-Nummer, übergeben um schon vorhandene Beiträge aus der Datenbank zu filtern und gegebenenfalls einen neuen Eintrag mit korrekten Attributen anzulegen.

Nun werden dem Benutzer alle Kommentare oder eben Errata die bereits für diese RFC-Zeile verfasst wurden jeweils mit Erstellungsdatum, Uhrzeit und Benutzernamen des Verfassers aufgelistet.

Ist der Benutzer registriert so kann er zusätzlich einen eigenen Beitrag erstellen. Hierfür wird ihm ein Feld für seinen Beitragstext, ein *Post comment/Post errata*-Button zum Absenden des Beitrags und ein *Reset*-Button zum leeren des Textfeldes angeboten. Der *Username* ist an dieser Stelle bereits automatisch mit seinem Benutzernamen gesetzt und kann daher nicht verfälscht werden. Nach Absendung des Beitrags wird der Nutzer über *HttpResponseRedirect("...")* direkt auf die aktualisierte Seite geleitet, in der nun auch sein neuer Beitrag sichtbar ist.
Ist der Besucher nicht registriert werden lediglich die schon verfassten Beiträge aufgelistet. Dass die Möglichkeit Beiträge selbst zu verfassen ihm also an dieser Stelle nicht zur Verfügung steht, haben wir über eine Sicherheitsabfrage *{% if user.is_authenticated %}* im jeweiligen template *comment.html* und *errata.html* sichergestellt.

In den views haben wir zuerst die benötigten Datenbank-Objekte aus der jeweiligen Tabelle mithilfe der übergebenen Variablen RFC und Zeilennummer gefiltert und in einer Variable *entries* abgelegt. Dann wird über eine *if request.method == "POST"* Abfrage entschieden ob der Benutzer bereits eine Eingabe getätigt hat oder nicht.

Ist ersteres der Fall und zusätzlich seine Eingabe gültig, was wir über *if comment_form.is_valid()* prüfen, so wird ein neues Datenbank-Objekt, *Comment* oder *Errata*, angelegt und anschließend in die jeweilige Tabelle der Datenbank gespeichert. Hierfür setzen wir das Attribut *username* über *request.user.username* automatisch auf den Benutzernamen des registrierten Nutzers.

Der Text des Beitrags, also das Attribut *text* wird über die *form* durch *comment_form.cleaned_data[„comment“]* oder *errata_form.cleaned_data[„errata“]* festgelegt. Die weiteren Attribute *line* und *rfc*, die noch zur Erstellung des Datenbank-Objekts benötigt werden, setzen wir über die anfangs übergebenen Variablen. Zusätzlich wird noch das Attribut *date* benötigt, welches wir in der *models.py-Datei* über *date = models.DateTimeField(auto_now_add=True)* automatisch generieren.

Anschließend wird über *"send_mail()"* eine e-mail an den Verfasser dieses RFCs mit dem Inhalt des Beitrages gesendet.

Im anderen Fall, wenn also *request.method == "POST"* nicht zutrifft, rendern wir *"comment.html / errata.html"* mit dem entsprechenden Kontext über *render_to_response().*

Um die Standardausgabe der *form* (das sind die Formular-Elemente um Beiträge zu verfassen), die normalerweise durch die Funktion *{ form.as_p }* in den templates erzeugt wird, an unsere Anforderungen anzupassen, haben wir diese Funktion durch eigenen Code ersetzt um zusätzliche html-Tags zur Darstellung einfügen zu können.

7.2 *Profil Seite:*

Des Weiteren haben wir eine Profil-Seite implementiert. Diese wird nach erfolgreichem Login-Prozess in der Menü-Leiste sichtbar. Hier kann der eingeloggte Benutzer seine bisherigen Beiträge, also Comments und/oder Errata, ansehen, bearbeiten und auch löschen.

Hierfür war es nicht notwendig eine weitere App zu erstellen, stattdessen haben wir die Dateien zur Erstellung der Profil-Seite in die Login-App integriert.

- Der Pythoncode zur Programmierung liegt in *views.py*, und umfasst die views *profile_view()*, *editComment()*, *deleteComment()* und entsprechend *editErrata()* und *deleteErrata()*.
- Im *templates*-Ordner haben wir die HTML-Dateien *profile.html*, *editcomment.html* und *editerrata.html* abgelegt.
- Um die Benutzereingaben aufzunehmen haben wir, genau wie in der *blog*-App auch, das Django-Modul *forms* verwendet. In der Datei *forms.py* haben wir hierfür zwei Klassen *class EditCommentForm(forms.Form)* und *class EditErrataForm (forms.Form)* implementiert.
- Aus der Datenbank wurden für die Profil-Seite die Tabellen der *blog*-App *Comment(models.Model)* und *Errata(models.Model)* benutzt.

Sobald der Benutzer in der Menü-Leiste auf die Option *Profile* klickt wird er auf die entsprechende url */login/profile/* weitergeleitet. Ein Django-Mechanismus ruft dadurch implizit die Funktion *profile_view()* auf. Damit dieser Mechanismus funktioniert haben wir in der *urls.py* jeweils ein Url-Funktions-Tupel in *urlpatterns* angelegt. In der *profile_view()* werden die zum Benutzer gehörenden Beitrags-Objekte aus der Datenbank gefiltert und als Kontext der *render_to_response()*-Funktion übergeben. Diese aktiviert dann die *profile.html*-Datei, welche die Beiträge des Benutzers in gesonderten Tabellen, eine für Comments und eine andere für Errata, mit zugehörigem RFC und Datum auflistet und jeweils Buttons *edit* und *delete* bereitstellt.

Möchte der Benutzer einen seiner Beiträge bearbeiten kann er dies über den Button *edit* realisieren. Durch ein *onClick*-Event wird intern die entsprechende view *editComment* beziehungsweise *editErrata* aufgerufen. Diese views benötigen die Beitrags-ID, welche wir über die url übergeben und mit *request.GET.get()* abholen.

Da der Benutzer diese natürlich nicht sehen soll, die ID aber für die weitere Verarbeitung benötigt wird, haben wir die Beitrags-ID zusätzlich in der *form* als *hidden input: <input type="hidden" name="errata_ID" value={{ errata_ID }}>* temporär abgespeichert. Diese ID wird benötigt, da beim Editieren des Beitrags nur das *Text*-Attribut des schon vorhandene Datenbank-Objekts bearbeitet und kein neues Objekt erstellt wird, und so über die ID jederzeit in der view auf das benötigte Datenbank-Objekt zugegriffen werden kann.

Nach Anklicken des *edit*-Button wird der Benuter auf die url */editComment/* oder analog */editErrata/* weitergeleitet und die views *editComment()* oder *editErrata()* aufgerufen. Hier steht ihm ein Textfeld das initial mit dem zu bearbeitenden Beitrag gefüllt ist, ein *Save-changes*-Button und ein *Reset*-Button zur Verfügung. Im Textfeld kann der Benutzer seinen Beitrag bearbeiten, mit *Reset* wieder herstellen und über *Save Changes* wird die Änderung übernommen und der Benutzer wird wieder zurück auf seine Profil-Seite geleitet.

Über die delete-Option, welche auf der Profil-Seite für jeden Beitrag verfügbar ist, wird der Beitrag und somit das entsprechende Datenbank-Objekt unwiderruflich und ohne Nachfrage gelöscht. Der Benutzer wird anschließend direkt über *HttpResponseRedirect("/login/profile/")* auf seine nun aktualisierte Profil-Seite zurückgeleitet.

8 *Durchsuchen der RFCs und Login-System*

8.1 *Durchsuchen der RFCs*

Unser erstes Ziel war es *hello world* auf einer blanken HTML-Seite auszugeben. Dafür starteten wir zunächst eine neue Django-Applikation mit dem Befehl *python manage.py startapp "name".* Django erzeugt dann automatisch eine Ordnerstruktur und die Standard Dateien. Als nächstes mussten wir die Datei *views.py* bearbeiten, indem wir ihr mitteilten welches Template sie rendern soll und mit welchem Inhalt, wie in unserem Beispiel: *hello world*. Als vorletzten Schritt mussten wir die URL-Konfiguration in der *urls.py* vornehmen. Diese legt fest welche Daten dem Nutzer präsentiert werden und zwar über die jeweilige URL die er anfordert. Zuletzt mussten wir eine *HTML*-Datei erstellen, die das Template darstellt. Das Template kann einfache *if-Abfragen*, *for-Schleifen* sowie Daten bearbeiten. Mithilfe dieses Bezeichners *{{ variable }}* wird eine Variable an das Template übergeben. Bei Aufruf des entsprechenden Links wurde uns nun *hello world* im Browser ausgegeben.

Als nächstes Ziel nahmen wir uns vor eine Suchmaske zu realisieren. Dazu mussten wir in der HTML Datei eine HTML-Form erstellen, die es dem Benutzer erlaubt Suchbegriffe einzugeben. Zusätzlich mussten wir eine Datei namens *forms.py* in der zuvor erstellten Applikation anlegen. Mithilfe dieser Datei überprüft Django die Eingaben des Benutzers auf Korrektheit. Zum Beispiel wird der Datentyp überprüft oder es werden mögliche Angriffe verhindert. In der *views.py* gibt es dafür eine einfache Abfrage *form.is_valid()* die einen boolschen Wert zurückgibt. Bei erfolgreicher Validation lässt sich dann anhand der Suchbegriffe in der Datenbank nach dem passenden RFC suchen. Da wir die RFCs in der Tabelle mit Titel und Nummer abgespeichert haben, hatten wir zunächst das Problem, dass wir entweder nur nach dem Titel oder nur nach der Nummer suchen konnten. Wir lösten das Problem mithilfe des sogenannten Q-Objekts. Damit lassen sich mehrere Datenbankanfragen verknüpfen. Als wir die nötigen Daten aus der Datenbank abgefragt hatten, mussten wir die Python-Liste nur noch an das Template schicken und dort mit einer *for-Schleife* alle Ergebnisse ausgeben.

8.2 Einzelanzeige eines RFC's (RFC-View-App)

Als nächste Applikation hatten wir die Einzelanzeige der RFCs geplant. Dazu war es nötig, dass die beiden Applikation untereinander Daten austauschen können. Der Benutzer musste von den Suchergebnissen der Such-App auf die Einzelanzeige geleitet werden. In der Such-App haben wir dafür hinter jedes Ergebnis einen Link gesetzt und diesem die ID der RFC angehängt. Über die Django-Datei *urls.py* im Projektordner lies sich nun über reguläre Ausdrücke festlegen wie Django die aufgerufene URL bearbeitet. Durch folgende Einstellung werden die Links der Suchergebnisse mit einer Methode der RFC-View-App verknüpft:

- *url(r'^comment/(?P<result_id>.+)/$', 'rfcview.views.rfcview')*

Anhand der *result_id* kann Django nun in der Datenbank das passende RFC heraussuchen und auslesen. Mithilfe der Python Funktion *readlines()* lasen wir den kompletten Inhalt der Datei aus und übergaben es an das Template. Dabei trat das Problem auf, dass der Browser die *whitespaces* ignorierte. Wir lösten das Problem indem wir mithilfe der Python Funktion *replace(" ", "\xa0")* alle Leerzeichen durch *\xa0* ersetzten, bevor wir die Liste mit dem Zeileninhalt an das Template schickten. Im Template selbst wurde dann mithilfe einer for-Schleife durch die Liste iteriert. Dabei wurden gleichzeitig die Zeilennummern und die Buttons für die Kommentar- und Erratafunktion eingefügt.

Wir wollten für die Kommentar- und Erratafunktion eine Popup Box realisieren. Da unser Wissen in Java-Script aber leider nicht ausreichte, suchten wir im Internet nach einer Lösung und fanden eine vorgefertigte Popup Box namens *Greybox* (*http://orangoo.com/labs/GreyBox/)*. Die Benutzung war einfach, wir mussten nur die Dateien in das Template importieren und konnten dann innerhalb des Templates einen Link mit dem Attribut *rel="gb_page_center[600, 445]" setzen*. Wenn man dann als Benutzer auf diesen Link klickt, öffnet sich ein Fenster mit der Größe 600x445 und es wird die nächste Applikation, für Kommentare und Errata, geladen.

8.3 *Loginsystem*

Für das Login-System wurde das von Django mitgelieferte „User authentication“ Modul verwendet. Es verwaltet Benutzer-Accounts, Gruppen, Rechte sowie cookie-basierte Benutzer-Sessions.

Zunächst mussten wir das Modul installieren. Hierfür haben wir die entsprechenden Änderungen in der *settings.py* unseres Django-Projekts vorgenommen, um die App zu aktivieren, und anschließend die Datenbank neu synchronisiert, sodass alle nötigen Tabellen erstellt wurden. Gleichzeitig wurden mit dem *syncdb* Befehl alle neuen *Field*-Objekte in Django erzeugt. Mit Hilfe der *Field*-Objekte wird die Datenbank mit allen Tabellen inklusive ihren Felder auf python Ebene *gemapped*, sodass sie als Schnittstelle zwischen Datenbank und Programmcode dienen. Somit wird der Zugriff auf Datensätze über vordefinierte Methoden, die alle gängien SQL-Abfragen abdecken, ermöglicht und erspart die direkte *raw*-Abfrage der Datenbank.

Im nächsten Schritt erstellten wir eine neue App *Login*, um die Funktionen zu programmieren und die nötigen HTML-Templates zu erzeugen. Für unser Login-System wurde ein Template für den Anmelde-Vorgang, auf dem der Besucher seine Benutzerdaten eingibt um sich anzumelden, erzeugt.

Des weiteren sollte es eine Möglichkeit geben, sich zu registrieren. Zusätzlich wurde eine Account-Leiste erstellt, die auf jeder einzelnen Seite unseres Projekts implementiert wurde, sodass der Besucher jederzeit einen Überblick über seinen Login-Status und sein Profil erhält und die Möglichkeit hat sich abzumelden.

Die Programmierung aller Funktionen des Login-Systems erfolgte in der entsprechenden *views.py* der von uns erzeugten Login-App.

Auf der Register-Seite kann der Besucher einen Account erstellen. Hierfür reicht die Eingabe eines Benutzernamens, eines Passworts und einer gültigen E-Mail Adresse. Die eingegebenen Daten werden, mit Klick auf den *Register*-Button, als *POST* Daten an die zugehörige Methode in der *views.py* der *Login*-App geschickt. Alle Methoden in der *views.py* besitzen einen vordefinierten Parameter *request* in dem alle übergebenen Daten gespeichert werden.

Über diese Variable lässt sich dann mit Hilfe von gegebenen Methoden auf die gewünschten Daten zugreifen. Somit werden die erhaltenen Register-Daten ausgelesen und können weiter verarbeitet werden.

Um einen eindeutigen Login zu gewährleisten und Sicherheitslücken vorzubeugen, werden alle Eingabedaten des Besuchers bei der Registrierung und bei der Anmeldung auf Richtigkeit überprüft. Hierfür wurden von uns *Form*-Objekte erzeugt, die alle Eigenschaften und Vorgaben der Felder festlegen. Anhand dieser Form-Objekte werden die Eingabedaten geprüft. Sind diese gültig, wird der neue Benutzer in der Datenbank angelegt.

Zur Verschlüsselung von Passwörtern wird der PBKDF2 *(Password-Based Key Derivation Function 2)* Algorithmus verwendet.

Meldet sich ein Besucher an, so werden seine Eingabedaten ebenfalls zuerst auf Richtigkeit überprüft. Sind die Eingaben gültig, wird für den Benutzer eine Session-ID erzeugt, die in seinen Cookies sowie in der Datenbank gespeichert werden. Anhand dieser *Session-ID* wird der Benutzer auf jeder Seite unseres Projekts neu „erkannt“ sodass er weiterhin angemeldet bleibt. Die *Session-ID* verfällt nach einer gewissen Zeit. Der Zeitpunkt wird in der Datenbank generiert und in *expire date* unter der entsprechenden *Session-ID* gespeichert.

9 *Hindernisse:*

Natürlich gibt es in einem größeren Projekt immer größere und kleinere Hindernisse, die wir aber alle so weit möglich gut meistern konnten. Als größtes Problem während unserer Projektarbeit müssen wir an dieser Stelle aber die permanenten Verbindungsprobleme zum Hochschulnetz an eben dieser aufführen. Diese haben die gemeinsame Arbeit am Projekt nicht nur deutlich erschwert, sie haben ein gemeinsames Arbeiten an vielen Tagen unmöglich gemacht. Selbstverständlich ist uns bewusst, dass auch die Ressourcen eines WLAN-Netzes nicht unbegrenzt sind, jedoch, und das möchten wir an dieser Stelle noch einmal verdeutlichen, haben wir lediglich Textdateien über das Netzwerk übertragen sowie eine Verbindung zu unserem Server per SSH-Protokoll aufgebaut, sonst nichts, und das sollte in einem modernen Hochschulnetz unserer Meinung nach störungsfrei funktionieren.

Als ein weiteres Hindernis möchten wir die schlechte Qualität des Open-Source-Projektes Etherpad Lite schildern. Der Quellcode dieses Tools ist sehr unübersichtlich und es ist uns bis heute nicht immer klar, woher bestimmte Objekte kommen, wie der Datenaustausch der einzelnen Module funktioniert und wie der genaue Ablauf der Methodenaufrufe stattfindet. Die Dokumentation des Quellcodes ist an vielen Stellen schlecht und zum Teil sogar nicht gegeben.

Weiter war der asynchrone Ablauf, bedingt durch die Verfahrensweise von Nodejs, hinderlich. Nodejs verwendet nur einen Prozess für mehrere Benutzer. Um die anderen Nutzer durch eine länger andauernde Anfrage eines Clients an den Server nicht zu blockieren, werden Methoden asynchron aufgerufen. Das heißt, dass das Ergebnis eines Methodenaufrufes noch nicht vorliegt, während das Programm im Quellcode aber bereits an einer Stelle angekommen ist, an welcher dieses Ergebnis benötigt wird.

10 *Fazit:*

Rückblickend kann festgestellt werden, dass unser gruppeninternes Ziel sehr gut umgesetzt und erreicht wurde. Wir arbeiteten mit großer Motivation und Interesse. Während unserer Arbeit ist uns aufgefallen, dass die Kommunikation eines der wichtigsten Elemente produktiven Arbeitens ist. Besonders zu betonen sind u.a. die tolle Zusammenarbeit und Hilfsbereitschaft im Team, sowie die Entwicklung vorgreifender Ideen für das Projekt.

11 Anhang:

11.1 Chronologischer Bericht:

14.03.2012: erstes Treffen der Gruppenmitglieder

-Besprechung des allgemeinen Vorgehens

-Terminfestlegung für das wöchentliche Treffen

04.04.2012: erstes Treffen mit Herr Winter

-Vorstellung aller Teilnehmer

-Einführung in das Projektthema

10.04.2012:

-Wiki erstellt

11.04.2012: wöchentliches Treffen

-Analyse der Problemstellung

-Arbeitsumgebung eingerichtet

15.04.2012 bis 21.04.2012:

-Recherche über Webentwicklung mit Python, WSGI, cronjobs, rsync

18.04.2012: wöchentliches Treffen

-Arbeitsumgebung eingerichtet

25.04.2012: wöchentliches Treffen

-allgemeine Konzeptvorstellung

29.04.2012 bis 05.05.2012:

-Konzept Feinschliff

Einteilung des Projekts in drei Schwerpunkte und Teams :

-Comment und Errata System (Sebastian Pröll, Hannah Walkiw)

-RFC Suchmaschine (Max Schleuniger, Rafael Schmitt)

-Draftbearbeitung über Etherpad (Andre Majeres, Sanim Rashid, Michael Jünger, DenisRichter)

09.05.2012: wöchentliches Treffen

-Detaillierte Konzeptvorstellung

23.05.2012: wöchentliches Treffen

-Präsentation der bisherigen Arbeiten und Besprechung

-Wahl der Präsentationsform für den Projekttag

-E-Mail wegen gewünschter Präsentationsform an Frau Matzke geschrieben

10.06.2012:

-Nach Lösungen für das VPN-Verbindungsproblem gesucht

-VPN-Verbindungsproblem unter Linux durch Änderung der VPN- Konfigurationen behoben

13.06.2012: wöchentliches Treffen

-Datenbankstruktur nach den Normalformen sauber überarbeitet

-Datenbank Model erstellt

14.06.2012 bis 19.06.2012:

-Datenbank Schritt für Schritt neu modelliert

-Wiki überarbeitet und aktualisiert

21.06.2012 bis 26.06.2012:

-Stand für den Projekttag organisiert

-Materialliste den für Projekttag erstellt und an Frau Matzke gesendet

27.06.2012 bis 02.06.2012:

-Plakat für den Projekttag erstellt

-Datenbank-Modellierung in Visio übertragen

-Dokumentation geschrieben

11.2 *Datenbankmodell:*