Projektbericht

Theresa Schüttig, Benjamin Hempel, Marc Michel, Mauritius Berger, Lukas Hirsch

14. August 2020

Inhaltsverzeichnis

Planung

Projektorganisation (BHe)

Team und Rollen

Backup-Rollen sind in eckigen Klammern angegeben.

Name	Kürzel	E-Mail	Rolle(n)
Mauritius Berger	MBe	s79149@htw-dresden.de	Projektleiter, [Developer]
Lukas Hirsch	LHi	s79199@htw-dresden.de	Tester, Technical Writer
Theresa Schüttig	TSc	s79136@htw-dresden.de	Developer, [Tester]
Benjamin Hempel	BHe	s79132@htw-dresden.de	Developer
Marc Michel	MMi	s77201@htw-dresden.de	Architekt, [Developer] (SE II)
Stefan Holland	-	-	Architekt, [Analyst] (SE I)
Lucie Jill Urbons	-	-	Analyst (SE I)
Helene Uhlig	-	-	Analyst (SE I)

Kommunikation

Innerhalb des Teams

Innerhalb des Teams haben wir zwei Kommunikationswerkzeuge benutzt.

Zur schnellen Kommunikation bei auftretenden Problemen oder Fragen sowie zum Vereinbaren von Treffen - sowohl persönlich als auch remote - haben wir eine WhatsApp-Gruppe erstellt. Wir haben uns für WhatsApp entschieden, da es bereits von allen Teammitgliedern vor dem Projekt verwendet wurde und damit eine hohe Erreichbarkeit gewährleistet.

Aufgrund der COVID-19-Pandemie waren wir gezwungen, nahezu alle Treffen in Software Engineering II remote abzuhalten. Zur Durchführung der Meetings haben wir einen eigenen Discord-Server aufgesetzt. Die Wahl fiel auf Discord, da es bereits von einigen Teammitgliedern vor dem Projekt verwendet wurde, umfangreiche Möglichkeiten zum Teilen des eigenen Bildschirms bereitstellt und eine gute Audioqualität bietet.

Mit den Stakeholdern

Die Kommunikation mit dem StuRa erfolgte primär ebenfalls über die oben erwähnte WhatsApp-Gruppe und den Discord-Server. Das hat den Vorteil, dass alle Teammitglieder die Kommunikation

in Echtzeit mitverfolgen konnten und es keinen "Mittelsmann" gab, welcher die Informationen anschließend an alle verteilen musste.

Vereinzelt wurden außerdem der StuRa-eigene Discord-Server sowie E-Mails als Kommunikationsmittel genutzt.

Eingesetzte Werkzeuge

Kommunikation

siehe Projektorganisation → Kommunikation

- WhatsApp
- Discord
- E-Mail

Planung

Trello

Trello ermöglicht es, Kanban-artige Boards zur Organisation von Aufgaben zu erstellen und gemeinsam in Echtzeit an diesen zu arbeiten. Wir haben das Tool verwendet, um den aktuellen Status jedes Work Items (z.B. offen, in Arbeit, Review benötigt, Tests benötigt, …) abzubilden und auf einen Blick ersichtlich zu machen. Für jedes Work Item gab es eine Karte, in welcher sich Details zu diesem befanden (z.B. Akzeptanzkriterien oder Screenshots bei Bugs).

Wir haben uns auf folgenden Gründen für Trello entschieden:

- zahlreiche Elemente zur Ausgestaltung von Work Items (Checklisten, Text, Bilder, …)
- Hinzufügen von Fristen und Teammitgliedern zu Work Items möglich
- Work Items können kommentiert werden; ermöglicht Diskussion unter Teammitgliedern
- Integration mit GitHub, um z.B. Commits, Issues oder Pull Requests zu einem Work Item hinzuzufügen
- Work Items können mit Labels (z.B. Bug, Feature, wichtig) versehen werden, um diese zu klassifizieren

Weitere

- GitHub Issues
- Essence Navigator

Dokumentation

siehe Durchführung → Hauptaktivitäten → Dokumentation

- Sphinx

- AsciiDoc
- Visual Paradigm
- diagrams.net

Durchführung

Implementierung (BHe)

Organisation und Scope der Work Items

Alle zu erledigenden Work Items wurden in unserem Trello-Board organisiert und zunächst der Spalte "Backlog" zugeordnet. Beim Hinzufügen eines neuen Work Items wurde diesem in der Regel bereits eine grobe Beschreibung der zu erfüllenden Kriterien hinzugefügt.

[work item beispiel] | *images/work_item_beispiel.png*

Figure 1. Beispiel eines Work-Items im Trello-Board

Zu Beginn umfasste ein Work Item in der Regel einen Use Case. Da ein Use Case in unserem Fall üblicherweise eine komplette CRUD-Funktionalität (hinzufügen, bearbeiten, ansehen, löschen) beinhaltet, waren die Work Items zu Beginn sehr aufwändig und zogen sich über 2-3 Iterationen bzw. 4-6 Wochen. Der Vorteil wiederum ist, dass es somit für jeden Use Case einen Experten gab, welcher bei Fragen zu diesem Use Case den anderen Teammitgliedern zur Verfügung stand. Dies wurde dadurch verstärkt, dass die Developer zuvor noch nicht mit Django gearbeitet haben und sich infolgedessen zunächst in das Framework einarbeiten mussten. Eine explizite Aufwandsschätzung, etwa in Form von Punkten oder Arbeitsstunden, wurde jedoch aufgrund des vergleichsweise kleinen Umfangs des Projekts nicht durchgeführt.

Der Umfang (Scope) der Work Items nahm im Verlauf der Implementierung stetig ab. Das liegt daran, dass nach Implementierung der Use Cases schon ein Großteil der Funktionalität vorhanden war. Später hinzugestoßene Work Items bezogen sich in der Regel auf gewünschte Verbesserungen seitens der Stakeholder, kleinere Anpassungen oder Bugfixes. Somit wurden zu Beginn 0,33-0,5 Work Items pro Iteration und Teammitglied bearbeitet, während es am Ende aufgrund des geringeren Scopes pro Item 2-3 waren.

Aufsetzen der Anwendungsstruktur

Zu Beginn des Semesters wurde das Grundgerüst der Anwendung, d.h. die Generierung der Django-Anwendung sowie der einzelnen Apps, durch ein Teammitglied aufgebaut. Außerdem wurde bereits eine erste Version des Datenbankmodells entsprechend des vorher erstellten Entwurfs implementiert. Weiterhin wurden bereits globale Abhängigkeiten wie die UI-Bibliothek Materialize und jQuery eingebunden und das Menü der Anwendung, bestehend aus Header und Footer, implementiert.

Erstellen von UI-Prototypen

Vor der Implementierung der Funktionalität wurden zunächst UI-Prototypen für alle Use Cases umgesetzt. Diese wurden genutzt, um früh in der Entwicklung Feedback zur Benutzerführung einzuholen und eventuelle Probleme oder fehlende bzw. von der Vorstellung der Stakeholder abweichende Funktionen zu identifizieren. Außerdem konnte dadurch schnell eine gemeinsame Designsprache für die Anwendung gefunden werden, welche in allen Bereichen der App Verwendung findet. Da die erstellten Prototypen von Anfang an nicht als Wegwerf-Prototypen geplant waren, wurde bereits auf eine möglichst hohe Qualität der Benutzererfahrung und des

Codes hingearbeitet.

Ablauf der Abarbeitung eines Work Items

Beginn der Iteration

Zu Beginn der Iteration wurden alle noch zu erledigenden Aufgaben im zweiwöchentlichen Iterationsmeeting betrachtet. Jeder Developer hatte anschließend die Möglichkeit, entsprechend der eigenen Interessen, Fähigkeiten und der verfügbaren Zeit die Work Items auszuwählen, welche er/sie erledigen möchte. Der Zeitaufwand pro Iteration und Developer war hierbei flexibel, es wurde aber darauf geachtet, dass jeder Developer summiert über alle Iterationen in etwa den gleichen Aufwand geleistet hat. Glücklicherweise kam es nicht dazu, dass Work Items nicht vergeben wurden oder ein Teammitglied sich weigerte, ein Item anzunehmen.

Hat ein Developer ein Work Item angenommen, wird dieses in Trello dem Mitglied zugewiesen und in die Spalte "Aktuelle Iteration" verschoben. Der Developer sieht sich außerdem noch im Meeting die bereits vorhandene Beschreibung sowie die Akzeptanzkriterien des Work Items an. Gegebenenfalls werden diese noch erweitert sowie eventuell aufgetretene Fragen geklärt.

Branching-Strategie und Implementierung

Zunächst wurde vom bearbeitenden Developer ein neuer Branch mit einem aussagekräftigen Namen angelegt, auf welchem anschließend die eigentliche Implementierung erfolgte. Bei der Implementierung selbst hatte jeder Developer freie Hand; bei Änderungen am Datenbankmodell oder der grundlegenden Struktur der Anwendung mussten aber alle anderen Teammitglieder informiert werden. Am Datenmodell mussten mehrmals Änderungen vorgenommen werden, insbesondere aufgrund von Feedback durch die Stakeholder (siehe Abschnitt unten). Diese gestalteten sich nicht immer einfach, da beispielsweise die Historie auf alle anderen Klassen zugreift und somit Inkonsistenzen in der Benennung oder dem Zugriff auf das Datenmodell entstanden sind.

In den Zwischenmeetings, welche immer etwa zur Mitte einer Iteration abgehalten wurden, wurde sich über den aktuellen Stand der in Bearbeitung befindlichen Work Items ausgetauscht. Fragen, Probleme und Unsicherheiten konnten so direkt im Meeting geklärt werden, etwa via geteiltem Bildschirm. Bei schwerwiegenden Problemen wurde vereinzelt auch Pair Programming durchgeführt oder ein anderer Developer zum betroffenen Work Item hinzugezogen.

Vor allem in späteren Iterationen wurde darauf geachtet, den Code direkt während der Implementierung ausführlich mittels Sphinx zu dokumentieren. Das hat den Vorteil, dass der bearbeitende Developer sowieso gerade im richtigen Kontext arbeitet und deshalb eine qualitativ hochwertigere Dokumentation entsteht. Außerdem vermeidet man dadurch, dass eventuell Teile des Codes nicht dokumentiert werden.

Erstellen einer Pull Request und Review-Prozess

Nachdem die Implementierung fertiggestellt und vom bearbeitenden Developer händisch auf ihre Grundfunktionalität geprüft wurde, erstellte dieser eine Pull Request für den zuvor angelegten Branch. Diese wurde anschließend mit dem jeweiligen Work Item im Trello-Board verknüpft und

die anderen Mitglieder beispielsweise per WhatsApp über das Anlegen der Pull Request informiert. Ein Reviewer wurde in dieser nicht festgelegt, da Pull Requests in der Regel von einem beliebigen Teammitglied mit freien Kapazitäten bearbeiten wurden. Außerdem wird das Work Item in Trello in die Spalte "Review benötigt" verschoben.

[pull request beispiel] | *images/pull_request_beispiel.png*

Figure 2. Beispiel für eine Pull-Request

Der Reviewer checkt nun den Branch, für welchen die Pull Request gestellt wurde, aus und startet den Entwicklungsserver. Anschließend werden die im Work Item umgesetzten Bestandteile gründlich händisch überprüft, um bereits eventuelle Bugs oder Verbesserungsvorschläge einzubringen. Es wird außerdem darauf geachtet, dass die Umsetzung der Designsprache entspricht und der Quellcode ordentlich geschrieben und organisiert ist. Besteht Nachbesserungsbedarf, so wird dies in der Pull Request vermerkt und der bearbeitende Developer darüber informiert. In späteren Iterationen war es außerdem Pflicht, dass die Pull Request alle Tests, welche im Rahmen einer Continuous Integration mittels GitHub Actions automatisch ausgeführt wurden, besteht. Sollte es hier zu Abweichungen kommen, wurde der Tester informiert und mit diesem gemeinsam die Ursache für das Fehlschlagen der Tests ermittelt.

Ist der Reviewer mit den Ergebnissen zufrieden, wird die Pull Request gemerged und das Work Item auf dem Trello-Board in die Spalte "Tests benötigt" verschoben. Im Anschluss kümmert sich nun der Tester darum, die umgesetzte Funktionalität entsprechend zu testen.

Ende der Iteration

Am Ende der Iteration wurde das Ergebnis der durch einen Developer abgeschlossenen Work Items den anderen Teammitgliedern vorgestellt. Dabei konnte auch erstes Feedback zur Implementierung eingeholt und ggf. Verbesserungsvorschläge eingebracht werden, noch bevor die Stakeholder das Ergebnis zu Gesicht bekamen.

Besonders bei den großen Work Items zu Beginn des Semesters, welche in der Regel einen ganzen Use Case umfassten, kam es häufig vor, dass dieses nicht in einer Iteration fertiggestellt werden konnte. Der/die bearbeitende Developer/in hatte die Verzögerung im Meeting zu begründen und das Work Item wurde in die nächste Iteration übernommen. Glücklicherweise kam es dadurch zu keinem Stillstand bei einem der Teammitglieder, es blieben also keine Work Items ewig unbearbeitet liegen.

Feedback von Stakeholdern

In regelmäßigen Abständen wurden die Stakeholder zu gemeinsamen Meetings eingeladen, um den aktuellen Fortschritt der Implementierung zu zeigen und Feedback einzuholen. Dabei wurden mittels geteiltem Bildschirm die seit dem letzten Treffen umgesetzten Funktionalitäten gezeigt und die Stakeholder um Verbesserungsvorschläge gebeten.

Besonders viel Feedback erhielten wir dabei zu Beginn der Entwicklung zu den UI-Prototypen. Insbesondere bei den Checklisten und der Historie sind hier am Ende der Analyse immer noch offene Fragen geblieben, da der Kunde diese beiden Bestandteile zunächst nicht für uns nachvollziehbar und vollständig beschreiben konnte. Anhand der Prototypen konnte das

Verständnis für das Zielbild dieser beiden Funktionalitäten geschärft werden und die gewünschten Änderungen noch vor der Implementierung der Anwendungslogik umgesetzt werden.

Ein zweiter großer Punkt war die Benennung der "Ämter" und "Referate". Wie sich etwa zur Mitte der Entwicklung herausstellte, waren diese Benennungen zu eng gefasst, da es auch Funktionen im StuRa gibt, welche eben keine Ämter sind und streng genommen auch nicht zu einem Referat gehören. Deshalb haben wir uns nach einiger Diskussion auf die Bezeichnungen "Funktion" und "Organisationseinheit" geeinigt. Das Problem hierbei war, dass dies zahlreiche Umbenennungen sowohl im User Interface als auch im Quellcode zur Folge hatte, welche leider aufgrund anderer Prioritäten bei der Entwicklung nicht vollständig konsistent umgesetzt werden konnten.

Insgesamt war diese Form des Feedbacks jedoch äußerst hilfreich und ermöglichte es uns, die Anwendung bestmöglich auf die Wünsche des Kunden anzupassen.

Test (LHi)

Im folgenden Abschnitt wird die Durchführung und Organisation der Tests der Software **StuRa-Mitgliederdatenbank** aufgeführt.

Organisation

Unser Testkonzept bestand im Groben aus fünf wesentlichen Bestandteilen und den Akzeptanztests durch den Kunden.

1. Grober Test der neu implementierten Funktion vor dem Mergen des Branches.
2. Erstellen von Unittests für die neue Funktion
3. Wenn möglich, Abdeckung eines Usecases durch einen automatisierten Test mit Selenium (Happy Path)
4. Refactoring der Tests (wenn möglich zu parametrisierbaren Funktionen)
5. Erstellen von Robustheitstests aus den Funktionen
6. Akzeptanztests

Wir haben uns für das Testframework Selenium entschieden für unsere UI-Tests, da dieses Framework auch in anderen Programmiersprachen verwendet werden kann und so lernen wir quasi nachhaltig. Weiterhin gab es sehr ausführliche Tutorials und eine sehr gute Dokumentation.

Die Priorität der Tests lag auf das Testen des Happy Path, da wir hier die Usecases abdecken konnten und somit erstmal die Basisfunktionalität der Software gewährleisten konnten. Außerdem wurden weitere Funktionen wie die Historie und die Pagination in den Tests passiv mit eingebaut und wurden somit nicht explizit getestet.

Das Ergebnis der Tests (ob alle Tests ohne Fehler durchgelaufen sind) wurde über eine extra dafür erstellte GitHub-Action in unserer README.md festgehalten. Vor jedem Merge wurden auch diese automatisierten Tests ausgeführt.

Der Kunde bekam außerdem schon 1 Monat vor dem eigentlichen Release eine Testinstanz von unserem Team bereitgestellt, in der er testen konnte wie er lustig war. Leider wurde es jedoch nicht so intensiv genutzt, wie wir uns es erhofft hatten.

Schwierigkeiten

Eine große Hürde bei den UI-Tests mit Selenium war der unterschätzte Arbeitsaufwand, diese Tests waren sehr zeitintensiv, und da der Tester mit beim Entwickeln kurzzeitig geholfen hat ist viel wertvolle Zeit verloren gegangen. Somit wurden die geplanten automatisierten Robustheitstests ausgelagert und nur teilweise manuell durch uns durchgeführt.

Es war allerdings auch anstrengend und zum Teil frustrierend, dass die UI-Tests bei der kleinsten UI-Änderung angepasst werden mussten. Wir hatten dieses Problem in unserem Team besprochen, und unsere Strategie dahingehend geändert, dass UI-Tests nur für fertige Gesammtmodule geschrieben werden. Somit konnten wir das Problem ein wenig in den Griff bekommen.

Übergabe (LHi)

Die Übergabe des entwickelten Systems lief ein wenig holprig ab, da wir leider vom StuRa keine Instanz auf ihrem Server bekommen hatten. Kurzerhand haben wir uns dafür entschieden die Software auf einen eigenem Server (Raspberry Pi 4) zu Deployen. Wir hatten uns dafür entschieden, weil wir dem Stakeholder auch mind. 1 Monat vor Abgabe schonmal eine Testinstanz geben wollten. Mit dem StuRa haben wir uns dann darauf geeinigt, dass wir ihnen trozdem noch für einen Zeitraum von 30 Tagen nach der Abnahme bei einem Deployment auf Ihrem Server helfen können. Außerdem haben wir uns dafür entschlossen, in einem Zeitraum von 90 Tagen einen Administrator und/oder Entwickler tiefergehend in die Software einzuarbeiten.

Dokumentation (LHi)

Unsere Dokumentation ist in 3 Bestandteile gegliedert.

1. Anforderungsanalyse
2. Projektbericht (Zusammenfassung, Reflexion)
3. Softwaredokumentation

Unsere Dokumentation ist ein iteratives Ergebnis unserer Arbeit. Wir haben zuerst die grundlegenden Informationen zum Softwaresystem dokumentiert und später, als wir noch ein wenig Zeit hatten, sind wir immer mehr ins Detail gegangen.

Anforderungsanalyse

Die Anforderungsanalyse ist im AsciiDoc Format geschrieben und wird auf dem Master-Branch automatisch durch eine GitHub Action erstellt, somit hat jeder auf dem Master-Branch die aktuellste Version.

In der Anforderungsanalyse stehen unsere grundsätzlichen Überlegungen für den Entwurf der StuRa-Mitgliederdatenbank.

Projektbericht

Der Projektbericht wurde auch im AsciiDoc Format geschrieben, da man hier andere AsciiDoc Dateien includieren kann. Somit wird das kollaborative Arbeiten mit Github gut unterstützt. Hier stehen unsere Nachüberlegungen zur StuRa-Mitgliederdatenbank und unsere Lesson-Learned.

Softwaredokumentation

Dieser Dokumentationabschnitt enthält:

- Benutzerdokumentation
- Administratordokumentation
- Entwicklerdokumentation
- Testdokumentation

Diese Dokumentation wird mit dem für Python entwickelten Dokumentationstool Sphinx erstellt. Die Entscheidung fiel auf Sphinx, da es aus dem Code heraus eine Dokumentation erzeugen kann und unser Coach hat uns dabei einen Tipp gegeben. Weiterhin können wir mit Sphinx die Dokumentation in html oder pdf erstellen. Die Erstellung zu html haben wir direkt genutzt um eine Github-Page für die Dokumentation zu erstellen. Damit der Kunde einen schnellen und einfachen Zugriff auf diese Dokumentation hat.

Die Dokumetation die nicht aus dem Code heraus erzeugt werden kann, wie z.B. die Benutzerdokumentation, haben wir im Dateiformat reStructuredText geschrieben, da sich dieses Dateiformat gut in Sphinx implementieren lässt.

Wesentliche Entscheidungen (BHe)

Python als Skript-/Programmiersprache

Wir haben uns für Python als Skript-/Programmiersprache entschieden. Hierfür gibt es mehrere Gründe: Zu einen wurde Python vom Kunden für eine eventuelle Weiterentwicklung bevorzugt, und zum anderen hatten einige Teammitglieder bereits Erfahrung mit Python. Weiterhin ist Python relativ leicht erlernbar, wenn man bereits mit anderen prozeduralen Programmiersprachen Erfahrung hat. Die im Vergleich zu anderen Sprachen geringere Performance war hierbei vernachlässigbar, da das Softwaresystem nur wenige gleichzeitige Nutzer bedienen muss.

Django als Web-Framework

Zur offiziellen Webseite

Wir haben uns für Django als Web-Framework entschieden. Hauptgrund für diese Entscheidung war die Popularität von Django, wodurch die Wahrscheinlichkeit hoch ist, dass bei einer Weiterentwicklung der Software bereits Erfahrung mit dem Framework besteht. Außerdem ist Django sehr gut dokumentiert und es existieren viele Anleitungen und eine große Community rund um das Framework.

Materialize als UI-Framework

Zur offiziellen Webseite

Wir haben uns für Materialize als Frontend-Framework entschieden. Der Hauptgrund hierfür ist, dass einige Teammitglieder bereits Erfahrungen mit Materialize haben. Außerdem ermöglicht das Material-Design ein modernes User Interface und lässt sich leicht auf eigene Bedürfnisse anpassen, z.B. was die eingesetzten Farben angeht. Es werden zahlreiche UI-Elemente, wie Formulare, Karten, Buttons und viele mehr mitgeliefert, die einfach eingesetzt werden können. Weiterhin ist Materialize gut dokumentiert und unabhängig von anderen Bibliotheken.

SQLite als Datenbanksystem

Wir haben uns für SQLite als Datenbanksystem entschieden. SQLite wird von Django standardmäßig als Datenbanksystem eingesetzt und ist für die Zwecke unseres Softwaresystems ausreichend, da zum einen die Daten aktuell nicht von anderen Systemen verwendet werden sollen und zum anderen die Performance-Einbußen bei wenigen gleichzeitigen Nutzern vernachlässigbar sind. SQLite hat den Vorteil, dass kein zusätzlicher Datenbank-Server aufgesetzt werden muss und die Daten durch die Ablage als Datei einfach zu portieren und zu sichern sind. Weiterhin fällt durch den nicht vorhandenen Server ein potenzieller Angriffsvektor weg. Sollten die Daten später auch in anderen Systemen benötigt werden, ist eine Portierung von SQLite zu anderen SQL-Datenbanksystemen einfach umzusetzen.

Verwendung der integrierten Admin-Oberfläche

Wir haben uns dazu entschieden, die in Django integrierte Admin-Oberfläche für einige Funktionalitäten unserer Anwendung zu benutzen. Bei diesen Funktionalitäten handelt es sich um Verwaltungsaufgaben, wie z.B. dem Hinzufügen neuer Systemnutzer, oder um Daten, die sich nur selten ändern und für die eine einfache CRUD-Funktionalität ausreicht. Dies betrifft z.B. die Organisationseinheiten, Unterbereiche und Funktionen des StuRa. Ein weiteres Kriterium war, dass die in der Admin-Oberfläche verwalteten Daten logischerweise nur von Administratoren bearbeitet werden müssen und können. Dadurch konnte unnötiger Implementierungsaufwand vermieden und somit mehr Zeit auf die Kernfunktionalitäten der Anwendung gelegt werden.

Sphinx zur Code- und Test-Dokumentation

Zur offiziellen Webseite

Wir haben uns für Sphinx zur Code- und Test-Dokumentation entschieden. Ein wichtiger Grund hierfür ist, dass man die Dokumentation sowohl als PDF- als auch als HTML-Dateien exportieren kann. Durch den Einsatz von reStructured Text (RST) sind außerdem zahlreiche Formatierungsoptionen gegeben und es ist möglich, die im Code eingebauten Docstrings in den Master-Dokumenten durch weitere Informationen zu ergänzen. Außerdem ist Sphinx sehr gut dokumentiert und wurde uns von unserem Coach empfohlen.

Ausschnitt aus der mit Sphinx und reStructured Text erstellten Code-Dokumentation

```
Views
~~~~~

.. automodule:: historie.views
    :members:
    :undoc-members:

Templates
~~~~~~~~~

Alle Templates sind unter `historie/templates/historie` zu finden.

list.html
""""""""""

Enthält den Grundaufbau der Historie. Die Historie wird hier in die 3 Tabs
"Mitglieder", "Ämter" und "Nutzer" unterteilt.
```

Verwendung von Asciidoc zur Dokumentation

Für die Dokumentation abseits von Code und Tests haben wir uns für Asciidoc entschieden. Wir haben uns explizit gegen das weiter verbreitete Markdown entschieden, da hier ohne weiteres keine "Hierarchie" an Dateien realisiert werden kann, d.h. man kann keine Markdown-Dateien in andere Markdown-Dateien inkludieren. Darüber hinaus haben wir Asciidoc bereits ausgiebig im

Praktikum und in Software Engineering I verwendet, wodurch alle Teammitglieder damit Erfahrung hatten. Auch sonst erfüllt Asciidoc unsere Anforderungen, da das Formatieren von Text und das Einfügen von Bildern einfach umsetzbar ist.

Ergebnisse

Reflexionen

Benjamin Hempel (BHe)

Insgesamt bin ich mit dem Projektverlauf zufrieden. Besonders stolz bin ich darauf, dass unser Softwaresystem vom Kunden so gut aufgenommen wurde und wahrscheinlich sogar in den Produktiveinsatz kommen wird. Ich bin auch stolz darauf, dass das Endprodukt bezüglich der Nutzererfahrung und des User Interface relativ hochwertig und sogar mobile-friendly geworden ist.

Im Rahmen des Projekts bin ich erstmals mit Python und dem Web-Framework Django in Berührung gekommen. Ich hatte damit die Möglichkeit (und auch die Aufgabe), eine neue Programmiersprache in Verbindung mit einem neuen Web-Framework zu erlernen, auch wenn mir das von Django eingesetzte MVC-Pattern bereits von Ruby on Rails bekannt war. Die größere Erfahrung für mich war jedoch, erstmals in einem größeren Team zu arbeiten. Bisher habe ich, sowohl auf Arbeit als auch privat, nur allein oder mit einer anderen Person zusammen an einem Projekt gearbeitet. Im Rahmen des Projekts konnte ich somit erstmals die Vorzüge (wie z.B. die Möglichkeit, von mehreren Personen Feedback oder Hilfe zu einem Problem erhalten zu können), als auch die Nachteile (wie z.B. Merge-Konflikte oder unterschiedliche Meinungen zu einem Thema) von Teamarbeit erleben.