UNIKASSEL
VERSITÄT

Fachbereich Elektrotechnik/Informatik
Fachgebiet Software Engineering

Toolgestütztes Event Storming für das Requirements Engineering

fulibWorkflows

BACHELORARBEIT

zur Erlangung des Grades eines Bachelor of Science
im Fachbereich Elektrotechnik/Informatik
der Universität Kassel

Eingereicht von:	Maximilian Freiherr von Künßberg
Anschrift:	Mönchebergstraße 31 34125 Kassel
Matrikelnummer:	33378673
Emailadresse:	maximilian-kuenssberg@uni-kassel.de
Vorgelegt im:	Fachgebiet Software Engineering
Gutachter:	Prof. Dr. Albert Zündorf Prof. Dr. Claude Draude
Betreuer:	M. Sc. Sebastian Copei
eingereicht am:	22. Februar 2022

Inhaltsverzeichnis

Listings

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

1 Einleitung

1.1 Motivation

TODO: Ach ja das Requirements Engineering in der agilen Softwareentwicklung ist und bleibt ein leidiges Thema. Dafür wurde von Alberto Brandolini das Event Storming ins Leben gerufen. Namensvetter Albert dachte sich: "Ja, geil - Hab ich Bock drauf." So begann er es zu erweitern und nun sind wir alle hier und schauen uns das an.

1.2 Ziele

TODO: Noch kurz warum Event Storming eine gute Idee ist und hilfreich in der Anforderungsanalyse sein kann. Natürlich schon mal kurz ansprechen, dass für Albert(o)s Event Storming eine Beschreibungssprache entwickelt wurde und man diese zum einfachen Bedienen mit einem Web-Editor versehen hat.

1.3 Methodik

TODO: Expertengespräch zum Prüfen der gesetzten Ziele. Was ist das und warum ist es für die Ziele ausreichend?

1.4 Existierende Konzepte

TODO: Oldschool alles auf papier -> keine mockups direkt mit framework Web Editoren wie Miro für das Erstellen von Boards.

1.5 Aufbau der Arbeit

TODO: Zuerst grundlegende Ideen und Technologien. Anschließend die Implementierung beschreiben. Evaluation mittels Expertengespräch mit Adam Malik. Fazit bezüglich Ziele und outcome. Ausblick für das Tool.

2 Grundlagen

Im folgenden Kapitel werden zuerst die grundlegenden Konzepte des *Event Stormings* erläutert. Hierbei wird auf dessen Herkunft und Entwicklung eingegangen. Neben diesen Grundlagen werden anschließend die für diese Arbeit notwendigen Änderungen und Erweiterungen dargelegt. Weiterhin werden die wichtigsten Technologien erläutert, welche für die Implementierung der Anwendungen nötig waren. Um eine bessere Übersicht zu schaffen, sind die Technologien nach dem Anwendungsteil, für welche diese nötig sind, unterteilt.

2.1 Event Storming

Dieses Unterkapitel befasst sich mit der Herkunft des *Event Stormings*, dem Domain-Driven Design (DDD). Zudem wird ein klassischer Ablauf eines *Event Stormings* erläutert und daran die Vorteile dieser Methodik für das Requirements Engineering beleuchtet. Abschließend werden die Änderungen und Erweiterungen, welche im Kontext dieser Arbeit vorgenommen wurden, erklärt.

2.1.1 Allgemein

TODO: Bevor ich dieses und das folgende Unterkapitel schreibe, erst noch mal das [Ver16] und [Bra21] lesen.

- Alberto Brandolini und das ES
- DDD als Grundlage
- Wie verläuft so ein ES Workshop (Beschrieben in seinem Buch, mehrfach)
- Wichtigsten Eckpunkte
- Warum ist es besser als Brain Storming oder ähnliches?

- Beispielhaftes Event Storming Board (Bild und beschreibungstext um später darauf bezug nehmen zu können)

2.1.2 Erweiterung

TODO: Hier das vorherige Kapitel abwarten um alle Änderungen/Erweiterungen besser daran fest zu machen.

- Erweiterungen für Wirtschaft (Pages -> daraus generierte Mockups, abgehen von dem "Wir wollen keinen PC benutzen"des ES)

- Ideen für die Lehre (Wird in dieser Arbeit nicht näher beleuchtet, da es für den Beleg der Funktionalität nicht mehr möglich ist dies ausreichend in der Bearbeitungszeit zu machen)

- ES -> Ablauf von Schritten -> Albert -> Workflow (Arbeitsablauf) beschreibungen -> Mögliche Idee zum besseren Nahebringen von komplexeren Abläufen in Vorlesungen. (Verbildlichung)

2.2 Technologien

Dieses Kapitel gibt einen Überblick über die verwendeten Technologien in der Umsetzung. Da die Implementierung aus verschiedenen Komponenten besteht, ist dieses Kapitel in drei weitere Unterkapitel aufgeteilt. Es wird somit getrennt auf die Java Library *fulib Workflows*, das Spring Boot Backend des *fulib Workflows Web-Editors* und das zum Editor dazugehörige Frontend, welches mit Angular umgesetzt wurde.

2.2.1 fulibWorkflows

fulib Workflows ist eine Java Library, welche Arbeitsabläufe, im Folgenden "workflows", in YAML Ain't Markup Language (YAML)-Syntax notiert als Eingabe nimmt und daraus sowohl ein Event Storming Board, im workflow beschriebene Mockups und Objekt-/ Klassendiagramme generiert. Welche Form die YAML-Eingabe haben muss und wie die Dateien aussehen und generiert werden, folgt in einem späteren Kapitel.

2.2.1.1 Antlr

Antlr bietet die Möglichkeit einen Parser über eine eigens geschriebene Grammatik zu generieren. Die Grammatik muss Links ableitend sein und ist in EBNF. TODO: Was bedeutet das? Der generierte Parser ermöglicht zudem das Aufbauen und Ablaufen eines *Parse trees*. Hierdurch ergibt sich die Möglichkeit während dem Parsen weitere Aktionen durchzuführen, welche den späteren Programmablauf eines Tools unterstützen können.

Listing 2.1: Beispiel einer einfachen Grammatik in Antlr

```
1 grammar AntlrExample;
2 prog:   (expr NEWLINE)* ;
3 expr:   expr ('*'|'/') expr
4 |       expr ('+'|'-') expr
5 |       INT
6 |       '(' expr ')'
7 ;
8 NEWLINE : [\r\n]+ ;
9 INT     : [0-9]+ ;
```

In Listing 2.1 ist ein Beispiel für eine einfache Grammatik zur Erkennung von mathematischen Gleichungen dargestellt.[Par14] Hierbei ist es lediglich möglich Zahlen mittels Klammern, Addition, Subtraktion, Multiplikation und Division miteinander zu kombinieren. Die Länge eines Ausdruckes ist durch den rekursiven Aufbau der Grammatik nicht begrenzt.

Listing 2.2: Einfacher mathematischer Ausdruck

```
1     ((199+2324)*43)/55
```

Die zuvor beschriebene Grammatik kann mittels weiterer Tools auf eine Eingabe geprüft werden. Eine zulässige Eingabe für die festgelegte Grammatik aus 2.1 ist in 2.2 dargestellt. Die Überprüfung auf die Richtigkeit einer Eingabe oder auch der Grammatik kann über Tools bereits vor einer Generierung von Code durchgeführt werden. Hierzu wurde das Diagramm aus Abbildung 2.1 mittels dem Antlr Plugin für IntelliJ generiert.

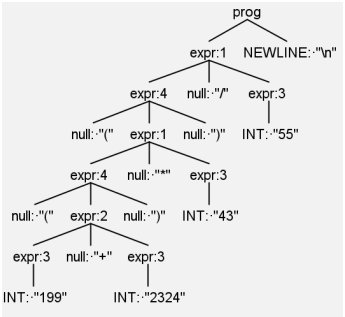

Abbildung 2.1: ParseTree für einen mathematischen Ausdruck

Hierbei ist ersichtlich, dass die Wurzel bei der obersten Regel **prog** beginnt und alle weiteren Kindknoten durch verschachtelte **expr** Regeln kreiert wurden. Der oberste Teilausdruck ist die Division aus einem komplexeren linken Ausdruck und dem rechten Ausdruck, welcher direkt einer Nummer zugeordnet werden konnte und somit keine weiteren Kindknoten mehr haben kann. Dies entsteht durch die Unterscheidung bei der Grammatik in Terminale und Nicht-Terminale. Hierbei werden wie in 2.1 dargestellt Nicht-Terminale Regeln kleingeschrieben und Terminale werden in Großbuchstaben verfasst. In der vereinfachten Grammatik sind lediglich ganze Zahlen als Eingabe erlaubt.

Dies sind lediglich die Grundlagen von Antlr, auf die genauere Verwendung des generierten Parsers und Besonderheiten der Grammatik wird in 3.1.1 eingegangen.

2.2.1.2 String Templates

StringTemplate gehören wir das vorherige Antlr zum *Antlr Project*. Antlr verwendet ebenfalls String Templates zur Generierung von formatiertem Text, im Folgenden als *Code* bezeichnet. Templates (übersetzt: Schablone) ermöglichen es zum Beispiel die feste Syntax einer Programmiersprache wie Java mit variablen Werten für Variablen, Klassen, Methoden, also den generellen Bausteinen einer Sprache zu füllen. Durch diese Funktionalität bieten sich String Templates sehr gut zu Generierung von Dateien an. Ursprünglich ist *StringTemplate* eine Java Library, jedoch wurden bereits Portierungen für C, Objective-C, JavaScript und Scala erstellt.

Die folgenden Erläuterungen beziehen sich auf die Java Library von *StringTemplate*, da diese in dieser Arbeit verwendet wurde. Die einfachste Möglichkeit für die Verwendung eines String Templates ist in Listing 2.3 zu sehen.

Listing 2.3: "Hello World!" - Beispiel mittels StringTemplate

```
1 import org.stringtemplate.v4.*;
2 ...
3 ST hello = new ST("Hello, <name>!");
4 hello.add("name", "World");
5 String output = hello.render();
6 System.out.println(output);
```

Die Klasse *ST* aus Zeile 3 kann mit einem String initialisiert werden. In diesem Beispiel wurden als Begrenzer für das zu ersetzende Stück des Textes <> verwendet. Im Anschluss wird dem neuen *ST* Objekt mithilfe der add()-Methode ein bestimmter Wert hinzugefügt. Der erste Parameter der Methode ist der Bezeichner innerhalb eines Templates, zu beachten ist die Angabe des Bezeichners ohne die Begrenzer. Der eigentliche Wert wird als zweiter Parameter übergeben und besitzt in diesem Beispiel den Text **World**. Um nun den fertig ersetzten Text aus dem Template und dem übergebenem Wert zu bekommen, muss auf

dem *ST* Objekt die Methode render() aufgerufen werden. Hierbei werden die Platzhalter des Templates durch den zuvor übergebenen Wert ersetzt und als String zurückgegeben. In Zeile 6 wird nun abschließend der fertige Text auf der Konsole ausgegeben, ‘Hello, World¡. Dieses Beispiel entstammt der offiziellen Webseite von *StringTemplate*.[Par13]

Für ein strukturiertes Arbeiten mit vielen Templates bietet *StringTemplate* die Möglichkeit StringTemplateGroups zu erstellen. Hierbei können mehrere Templates in einer Datei beschrieben werden um aufeinander aufbauende Templates nicht im Code, sondern einer gesonderten Datei zu organisieren. In diesen Dateien, welche die Dateiendung **.stg** tragen, können die Begrenzer (eng.: Delimiters) frei gewählt werden. Dies ist je nach Kontext des Templates nötig, da zum Beispiel die Generierung von HTML-Dateien, welche <> als Zeichen zum Abgrenzen von Bereichen verwenden. Bei der Wahl der Begrenzer sollte somit stets auf die Wahl der Zeichen im Kontext der zu generierenden Sprache geachtet werden. Zum Parsen einer StringTemplateGroup wurde ein mit Antlr generierter Parser verwendet.[Ter15]

Listing 2.4: Beispiel einer .stg-Datei

```
1 delimiters "{", "}"
2
3 example(topic, language) ::= <<
4 <span>
5     This test about {topic} is written in {language}.
6 </span>
7 >>
```

Wie zuvor beschrieben ist in Listing 2.4 zu erkennen, dass in Zeile 1 die Begrenzer auf gesetzt wurden. Dies hat den Hintergrund, dass in diesem Beispiel ein Text in eine HTML-Datei generiert werden soll. Hierfür könnten auch die Standardbegrenzer verwendet werden, allerdings müsste dann anstelle von <**span**> **span** stehen. Da dies für HTML-Dateien allerdings einen immensen Aufwand bedeutet, macht die Nutzung anderer Begrenzer Sinn. In Zeile 3 werden für ein StringTemplate, sowohl der Name des Templates, als auch Übergabeparameter definiert werden. Ein StringTemplate wird durch » geschlossen und das nächste Template könnte definiert werden. Die Begrenzer in Zeile 5 zeigen, dass alles, was sich zwischen Ihnen befindet, einen Übergabeparameter in sich trägt. Somit ist das Wiederverwenden des Templates und die variable Befüllung gewährleistet.

Um diese Templates nun in einem Java Programm zu verwenden, benötigt es unter anderem die zuvor beschriebenen ST Klasse, sowie die Klasse *STGroupFile*, welche für die Verwaltung der stg-Datei als auch deren Templates benötigt wird. In Zeile 6 von Listing 2.5 ist zu erkennen, dass einem STGroupFile Objekt bei der Initialisierung eine URL übergeben werden muss. Diese URL verweist auf die stg-Datei. Im Anschluss kann, wie in Zeile 8 ersichtlich, über die getInstanceOf()-Methode auf ein bestimmtes Template in der stg-Datei zugegriffen werden. Hierbei ist es wichtig, keine Fehler bei der Benennung zu

machen. Schließlich ist die weiterführende Verwendung bereits zuvor mittels der ST-Klasse beschrieben worden.

Listing 2.5: Nutzung einer STG-Datei in Java

```
1 import org.stringtemplate.v4.ST;
2 import org.stringtemplate.v4.STGroupFile;
3 ...
4
5 URL resourceUrl = Class.class.getResource("Example.stg");
6 STGroupFile exampleGroup = new STGroupFile(resourceUrl);
7
8 ST st = exampleGroup.getInstanceOf("example");
9 st.add("topic", "the university");
10 st.add("language", "english");
11
12 String output = st.render();
13 System.out.println(output);
```

Bei der Ausführung dieses Beispieles würde auf der Konsole der Text aus Listing 2.2.1.2 angezeigt werden.

```
1 <span>
2     This test about the university is written in english.
3 </span>
```

2.2.1.3 JSON-Schema

JSON-Schemas sind Schemata, welche den Inhalt einer JSON-/YAML-Datei begrenzen können. Hierdurch ist es möglich, den Nutzer in seinen Eingaben zu begrenzen und bereits während dem Schreiben einer Datei dabei zu unterstützen sinnvolle Eingaben zu erstellen. In dieser Arbeit wird lediglich auf die Nutzung der Schema Version 7, die neuste Version, eingegangen, da diese in der Anwendung verwendet wurde.

JSON Schemas können Objektstrukturen in beliebiger Tiefe schachteln. Im folgenden Abschnitt werden die grundlegenden Elemente eines JSON Schemas erläutert. Weiterführende Funktionalitäten werden anhand der Implementierung in einem späteren Kapitel näher beleuchtet.

Ein einzelnes Objekt kann zur Verbesserung der späteren Nutzung mit einem Titel und einer kurzen Beschreibung versehen werden. Diese sind in Listing 2.6 in Zeile 2 und 3 dargestellt. *title* und *description* dienen lediglich der Nutzbarkeit für den Entwickler.

Listing 2.6: Objekt Beispiel eines JSON Schemas

```
1 {
2   "title": "Product",
3   "description": "A product from Acme's catalog",
4   "type": "object",
5   "properties": {
6     "productId": {
7       "description": "The unique identifier for a product",
8       "type": "integer"
9     }
10  },
11  "required": [
12    "productId"
13  ]
14 }
```

Einem Element muss stets ein *type*, also ein Typ, zugeordnet werden. Dies kann entweder ein Objekt, Zeile 4 in Listing 2.6, sein. Alternativ kann ein Element auch als Liste typisiert werden, dies wird in einem folgenden Beispiel genauer erläutert. Einem Objekt können nun *properties* hinzugefügt werden. Diese besitzen neben einem eindeutigen Bezeichner ebenfalls eine Beschreibung und ein Typ. Auf dieser Ebene kann der Typ eine Nummer, *integer* in Zeile 8, oder auch ein Text, welches den Typ *string* bekommen würde, sein. Ist eine der *Properties* ein notwendiges Feld, kann dies mittels dem Keyword *required* realisiert werden. Hierbei wird eine Liste an Bezeichnern hinterlegt, welche dem Objekt bereits zugeordnet wurden und somit stets vorhanden sein müssen. Sollten einem Objekt keine weitere properties hinzugefügt werden dürfen, ist dies mit dem Ausdruck aus Listing 2.7 möglich.

Listing 2.7: Begrenzung der Properties eines Schemas

```
1 "additionalProperties": false
```

Wie zuvor bereits beschrieben, kann ein Element auch als Liste deklariert werden. Dies ist an einem kleinen Beispiel aus Listing 2.8 dargestellt. Hierbei ist es möglich die *items* einer Liste genauer zu definieren. In diesem Beispiel müssen die Elemente einer Liste dem Schema aus dem Beispiel aus Listing 2.6 entsprechen.

Eine JSON-/YAML-Datei, welchem dieses Schema zugrunde liegt, besteht somit aus einer Liste an Produkten. Durch die Verwendung des *oneOf* Operators in Zeile 6, werden nur Elemente mit dem darunterliegenden Schema akzeptiert. Bei mehreren Einträgen in der *items* Aufzählung muss immer eines dieser Elemente auf das Objekt in der JSON-/YAML-Datei zutreffen.

Listing 2.8: Listen Beispiel eines JSON Schemas

```
1 {
2   "title": "Products",
```

```
3   "description": "A list of products from Acme's catalog",
4   "type": "array",
5   "items": {
6     "oneOf": [
7       {
8         "type": "object",
9         "properties": {
10          "productId": {
11            "description": "The unique identifier for a product",
12            "type": "integer"
13          }
14        },
15        "required": [
16          "productId"
17        ]
18      }
19    ]
20  }
21 }
```

Durch ein fest definiertes Schema ist es vielen IDEs, darunter auch IntelliJ und VSCode, welche aus einem Schema nicht nur die fertige Datei auf Fehler überprüfen, sondern dem Entwickelnden bereits zum Zeitpunkt des Schreibens einer Datei mittels Autovervollständigung unterstützen kann. Hierfür ist es möglich bereits erstellte JSON-Schemas im *SchemaStore* bereitzustellen. Dies ist eine zentrale Stelle um JSON-Schemas für IDEs bereitzustellen. Bei dem *SchemaStore* handelt es sich um ein Open-Source-Projekt, bei welchem die Einbringung eines neuen Schemas simpel gestaltet ist. Es ist möglich ein fertiges Schema fest dort zu hinterlegen, hierdurch muss für jede neue Änderung allerdings einen neuen Betrag erstellt werden. Dieser bedarf einer Zustimmung einem der Verwalter des *Schema Stores*. Da dies stets mit einer zeitlichen Verzögerung passiert, ist es möglich eine Verlinkung zu einem Schema zu erstellen. Somit ist es möglich Änderungen an einem Schema durchzuführen und diese Änderungen nach dem Hochladen direkt zur Verfügung stellen zu können, ohne weitere Schritte durchführen zu müssen. Zum aktuellen Zeitpunkt existieren 439 Schemas, welche durch *SchemaStore.org* für diverse IDEs bereitgestellt werden.[Kri21] Eine Liste aller IDEs, welche diesen Support mittels *schemastore* unterstützen sind unter folgendem Link zu finden: `https://www.schemastore.org/json/#editors`

2.2.1.4 fulibTools

TODO: Dank fulibTools ist auch fulib mit drin. FulibTools ist zur Generierung von Objektidiagrammen genutzt worden. Fulib (Bei FulibTools mit integriert) zur Generierung von Klassendiagrammen.

2.2.2 fulibWorkflows Web-Editor FE

TODO: Da brauchte es etwas mehr als beim BE. Die Entscheidungen für die verwendeten Technologien im Frontend wurden aufgrund der Idee der Integrierung vom Editor auf fulib.org getroffen.

2.2.2.1 Angular

TODO: Wir kennen es. Wir lieben es.

2.2.2.2 Bootstrap

TODO: Alles für den Dackel, alles für den Club unser Leben für die schön gestylten FEs. Simpel, oder? Ja, okay. Man nutzt dann auch ng-bootstrap für Angular Anwendungen. Natürlich auch noch bootstrap-icons. Will ja niemand traurig machen und von Adrian verdroschen werden.

2.2.2.3 Codemirror

TODO: Schönes Ding. Ngx-codemirror ist es dann speziell für eine Angular Anwendung geworden. Eigentlich alles out of the box benutzt.

2.2.2.4 Angular-split

TODO: Find ich schon gut zu erwähnen. Ohne die geile dependency wäre das FE nie so pornös geworden.

2.2.2.5 file-saver

TODO: Weitere erwähnenswerte dependency. Wird genutzt um Dateien, die vom BE kommen, auch herunterladen zu können.

2.2.2.6 ajv

2.2.2.7 js-yaml

2.2.3 fulibWorkflows Web-Editor BE

TODO: Yeey alle Technologien die ich im Backend benutzt habe.

2.2.3.1 Spring Boot

TODO: Framework mit dem man easy mal ein backend generiert bekommt. Durch Java und viele Dependency allerdings alles andere als ein leichtgewicht.

Dennoch musste ein Java backend her, da sonst fulibWorkflows nicht hätte integriert werden können. Jedenfalls nicht ohne noch mehr middle ware.

Zudem hatte ich im Praktikum mit Spring Boot Erfahrungen gesammelt. Die Verwendung von Annotations und dem aufsplitten zwischen Controller und Service ist mir bereits durch Nest.js bekannt gewesen.

Dennoch muss man sagen, dass durch die von Spring Boot bereits integrierten libraries nichts weiter außer fulibWorkflows hinzugefügt werden musste. Immerhin umfasst der Code vom Backend vielleicht 300 Lines of Code.

2.2.3.2 fulibWorkflows

TODO: Sollte mindestens irgendwo erwähnt werden. Kann man ggf. auch in dem Text zum Kapitel schreiben.

2.2.4 Deployment

TODO: Ein Web Editor will natürlich für alle erreichbar sein. Und fulibWorkflows muss auch irgendwo bereitgestellt werden, damit es das Backend und alle anderen interessierten benutzen können.

2.2.4.1 MavenCentral

TODO: MavenCentral ein wirklicher Hussarones was das publishen angeht. Glücklicherweise ist fulibWorkflows Teil der Fujaba Tool Suite, wodurch die benötigten Zugriffsrechte bereits vorhanden und andere Libraries bereits gepublished wurden. Hierdurch war es recht schnell möglich mit dem zuvor erworbenem Wissen fulibWorkflows zu publishen.

2.2.4.2 Heroku

TODO: Der Web-Editor soll immer erreichbar sein. Dies ist durch Heroku nur bedingt möglich. Heroku bietet allerlei Möglichkeiten verschiedenste Anwendungen bereitzustellen. Auch mit einem kostenlosen Plan ist es ohne Probleme möglich solch kleine Anwendungen bereitzustellen.

FE Deployment war easy, auch wenn ich erstmal wieder in eins meiner früheren Projekte gucken musste. BE Deployment war kniffliger, doch man ist nie der erste der eine Spring Boot application auf Heroku deployen will. Daher Tutorial reingefahren und ab ging der gebutterte Lachs.

3 Implementierung

TODO: Hier bin ich mir tatsächlich unsicher was alles reinsoll. Ich teile es auch erstmal in drei auf.

3.1 fulibWorkflows

TODO: Was gibt es hier so interessantes zu sehen? Gehen Sie weiter.

3.1.1 fulibWorkflows Grammatik

TODO: Beschreiben der ANTLR4 Grammatik für fulibWorkflows

Und natürlich auch wie mir das beim parsen der yaml geholfen hat. Dat wird ne lange Sektion.

3.1.2 Generierung dank fulibTools

TODO: FulibTools gibt gute Anbindung an Graphviz, wenn man sowieso schon mit fulib arbeitet.

3.1.3 schema

TODO: Da hab ich lange dran gesessen. Und ich glaube, selbst jetzt ist es kein gutes Schema. Allerdings tut es was es soll.

3.1.3.1 Mockups

TODO: Eigenes Datenmodell gebaut. Daraus die wichtigsten Infos gezogen. Dank String-Templates von antlr richtig easy zu bauen. Gilt für Html als auch Fxml.

3.2 editor-frontend

TODO: Alles was es zum FE so gibt.

3.2.1 iframes

TODO: Naja der editor basiert einfach darauf, dass es iframes gibt. Könnte man auch weg lassen?

3.2.2 codemirror

TODO: Eingerichtet und los ging es. Noch einen eigenes kleines Hint Add on geschrieben. Feddig. Musste es erstmal simpel halten. Gibt noch genügend zukünftige Ideen.

3.2.3 angular split

TODO: Danke an Adrian, der mit dazu geraten hat.

Nachdem ich mit purem css da grids erstellt habe, stand ich vor dem Problem der Veränderung von Größen. Angular split löst das Problem auf eine wunderbare Art und Weise.

Die dreiteilung der View war damit wirklich einfach. Auch wenn man aufpassen musste beim Verändern der größen, wenn man iframes benutzt. Da musste ein kleiner Fix rein, der aber auch bereits von den machern vorgegeben war.

3.3 editor-backend

TODO: Das wird wieder kurz.

3.3.1 Spring booterino

TODO: Gabelstablinski hier drüben war dank ein paar Annotations schnell erstellt und macht bisher keine Probleme.

Die zip Datei wird im BE zusammengebaut. Dafür gibt es schon alles fertig in java.util.zip. Da hab ich nach gegoogelt und alles lief wie von selbst.

Na gut, ich muss zugeben, dass die zip vom BE ans FE schicken und dann richtig runterladen können, ohne das mir alles um die Ohren fliegt schon einen Arbeitstag gebraucht hat. Manchmal muss man lediglich seine Dummheit für einen Moment ablegen.

4 Evaluation

TODO: Eigentlichen Anwendugszweck beschreiben

Event Storming nachvollziehbar für alle machen. Htmls einfach noch mal öffnen.

Besser mit Kunden über das Layout sprechen können -> Mockups mit grundlegendem Styling Kunden können einen Workflow mal durchklicken -> Next prev page

4.1 Expertengespräch

TODO: Gespräch mit Adam. Fragenkatalog erstellen. Alles aufzeichnen, wenn Adam zustimmt auch Bild und Ton -> Youtube bereitstellen nicht gelistet

4.2 Auswertung

Was ist die Meinung des Experten zu der Anwendung?

Kann ihm das helfen? (Begründung durch seine Vorerfahrungen)

5 Fazit

Machen die Erweiterungen Sinn?

Kann der Editor von Leuten verwendet werden, die nur wenig Programmier erfahrung haben?

Hilft es beim RE in der Wirtschaft?

Füllt diese Anwendung eine bestehende Lücke im RE? Wurde das zuvor gesetzte Ziel erreicht? Ist die grundlegende Idee gut, aber die Umsetzung nicht ausreichend durchdacht?

Alberts Drang immer wieder neues zu haben xD

6 Ausblick

Was hatten die Leute zu meckern und sollte daher umgesetzt werden?

Was ist mir selbst an Ideen gekommen?

fulibWorkflows ist Teil von Fulib und sollte daher auch über fulib.org erreichbar sein. Alles zusammen an einem Ort macht mehr Sinn als alles verstreut zu haben.

7 Quellenverzeichnis

[Par13] Terence Parr. *StringTemplate*. 2013. URL: https://www.stringtemplate.org/.

[Par14] Terence Parr. *Antlr*. 2014. URL: https://www.antlr.org/.

[Ter15] Gerald Rosenberg Terence Parr. *StringTemplateGroupGrammar*. 2015. URL: https://github.com/antlr/grammars-v4/blob/master/stringtemplate/STGParser.g4.

[Ver16] Vaughn Vernon. *Domain-Driven Design Distilled*. Addison-Wesley, 2016. ISBN: 978-0-13-443442-1.

[Bra21] Alberto Brandolini. *Introducing Event Storming*. 2021.

[Kri21] Mads Kristensen. *JSON Schema Store*. 2021. URL: https://www.schemastore.org/json/.