

Toolgestütztes Event Storming für das Requirements Engineering

fulibWorkflows

B A C H E L O R A R B E I T

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

Kurzfassung

TODO: This

Inhaltsverzeichnis

Kurzfassung	I
Inhaltsverzeichnis	II
Listings	IV
Abbildungsverzeichnis	V
Abkürzungsverzeichnis	VI
1 Einleitung	1
1.1 Motivation	1
1.2 Ziele	1
1.3 Methodik	1
1.4 Existierende Konzepte	1
1.5 Aufbau der Arbeit	2
2 Grundlagen	3
2.1 Event Storming	3
2.1.1 Allgemein	3
2.1.2 Erweiterung	4
2.2 Technologien	4
2.2.1 fulibWorkflows	4
2.2.2 fulibWorkflows Web-Editor Frontend	11
2.2.3 fulibWorkflows Web-Editor Backend	13
2.2.4 Deployment	14
3 Implementierung	16
3.1 fulibWorkflows	16
3.1.1 Workflow Format	16
3.1.2 JSON-Schema	20
3.1.3 Antlr Grammatik	22
3.1.4 Generierung von Dateien	26
3.2 fulibWorkflows Web-Editor Frontend	33
3.2.1 Code-Editor	34
3.2.2 Darstellung generierter Dateien	37

3.3	fulibWorkflows Web-Editor Backend	39
4	Evaluation	41
4.1	Expertengespräch	41
4.2	Auswertung	41
5	Fazit	42
6	Ausblick	43
7	Quellenverzeichnis	44
8	Anhang	A
8.1	fulibWorkflows JSON-Schema	A

Listings

1	Beispiel einer einfachen Grammatik in Antlr	5
2	Einfacher mathematischer Ausdruck	5
3	“Hello World!” - Beispiel mittels StringTemplate	6
4	Beispiel einer .stg-Datei	7
5	Nutzung einer STG-Datei in Java	8
6	STG Ausgabe auf Konsole	8
7	Objekt Beispiel eines JSON Schemas	9
8	Begrenzung der Properties eines Schemas	9
9	Listen Beispiel eines JSON Schemas	10
10	Beispiel aller vorhandenen “Post-Its”	17
11	Referenzieren eines anderen Schemas	21
12	Grammatik für Workflows	23
13	Grammatik für Notes	23
14	Schlüsselwörter zum Identifizieren von Notes	23
15	Grammatik von Attributen	24
16	Grammatik von Werten	24
17	Grammatik einer Page	24
18	exitPage-Methode	26
19	FulibYaml.stg	29
20	Generierungsmethode eines Objektdiagramms	30
21	Beispiel eines richtigen Data Notes	30
22	Schritte zum Aufbau eines Klassenmodells	31
23	Codemirror Konfiguration	35
24	Modell der vom Backend empfangenen Daten	37
25	Bereitstellung einer Methode	38
26	Definition der Endpunkte	39
27	Komplettes JSON-Schema	B
28	Page JSON-Schema	C
29	fulibWorkflows Antlr Grammatik	D
30	Codemirror Add-on für Vervollständigung	E

Abbildungsverzeichnis

2.1	ParseTree für einen mathematischen Ausdruck	5
2.2	Spring Initializr für eine Web Anwendung	13
3.1	Fehleranzeige in IntelliJ	22
3.2	Autovervollständigung in IntelliJ	22
3.3	Klassendiagramm fulibWorkflows	25
3.4	Mittels fulibWorkflows generiertes Event Storming Board	27
3.5	Mittels fulibWorkflows generierte Mockups	28
3.6	Mittels fulibWorkflows generierte Objektdiagramme	31
3.7	Mittels fulibWorkflows generiertes Klassendiagramm	32
3.8	FulibWorkflows Web-Editor Oberfläche	33
3.9	Download Fenster	34
3.10	Themes des Codemirrors	35
3.11	Autovervollständigung	36
3.12	Validierungerror als Toast	37
3.13	Event Storming Board in einem IFrame	38
3.14	Inhalt eines heruntergeladenen Zip-Archivs	40

Abkürzungsverzeichnis

CPaaS Cloud Platform as a Service

DDD Domain-Driven Design

DSL Domain Specific Language

EBNF Erweiterte Backus-Naur-Form

SDK Software Development Kits

ST String Template

STG String Template Group

YAML YAML Ain't Markup Language

1 Einleitung

TODO: this

1.1 Motivation

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

1.4 Existierende Konzepte

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

1.5 Aufbau der Arbeit

TODO: **this** 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 Erweiterte Backus-Naur-Form (EBNF). 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.

```
grammar AntlrExample;
prog:      (expr NEWLINE)* ;
expr:      expr ( '*' | '/' ) expr
|          expr ( '+' | '-' ) expr
|          INT
|          '(' expr ')'
;
NEWLINE   : [\r\n]+ ;
INT       : [0-9]+ ;
```

Listing 1: Beispiel einer einfachen Grammatik in Antlr

In Listing 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 Ausdrucks ist durch den rekursiven Aufbau der Grammatik nicht begrenzt.

$((199+2324)*43)/55$

Listing 2: Einfacher mathematischer Ausdruck

Die zuvor beschriebene Grammatik kann mittels weiterer Tools auf eine Eingabe geprüft werden. Eine zulässige Eingabe für die festgelegte Grammatik aus 1 ist in 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.

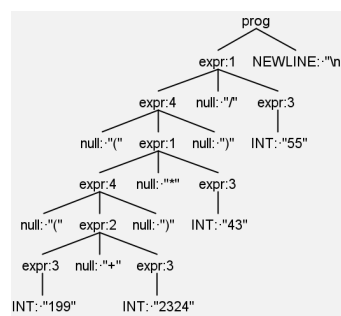


Abbildung 2.1: ParseTree für einen mathematischen Ausdruck

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 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 3.1.3 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 3 zu sehen.

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

Listing 3: "Hello World!" - Beispiel mittels StringTemplate

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, Worldj. 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.[\[Ter15\]](#)

```
delimiters "{", "}"
```

```
example(topic, language) ::= <<
<span>
    This test about {topic} is written in {language}.
</span>
>>
```

Listing 4: Beispiel einer `.stg`-Datei

Wie zuvor beschrieben ist in Listing 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` 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 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.

```
import org.stringtemplate.v4.ST;
import org.stringtemplate.v4.STGroupFile;
...

URL resourceUrl = Class.class.getResource("Example.stg");
STGroupFile exampleGroup = new STGroupFile(resourceUrl);

ST st = exampleGroup.getInstanceOf("example");
st.add("topic", "the university");
st.add("language", "english");

String output = st.render();
System.out.println(output);
```

Listing 5: Nutzung einer STG-Datei in Java

Bei der Ausführung dieses Beispiels würde auf der Konsole der Text aus Listing 6 angezeigt werden.

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

Listing 6: STG Ausgabe auf Konsole

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 7 in Zeile 2 und 3 dargestellt. *title* und *description* dienen lediglich der Nutzbarkeit für den Entwickler.

```
{
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    }
  },
  "required": [
    "productId"
  ]
}
```

Listing 7: Objekt Beispiel eines JSON Schemas

Einem Element muss stets ein *type*, also ein Typ, zugeordnet werden. Dies kann entweder ein Objekt, Zeile 4 in Listing 7, 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 des Schlüsselwortes *required* realisiert werden. Hierbei wird eine Liste an Bezeichnern hinterlegt, welche dem Objekt bereits zugeordnet wurden und somit stets vorhanden sein müssen. Das Beispiel stammt von der offiziellen JSON Schema Webseite.[Tea21c] Sollten einem Objekt keine weitere *properties* hinzugefügt werden dürfen, ist dies mit dem Ausdruck aus Listing 8 möglich.

```
"additionalProperties": false
```

Listing 8: Begrenzung der Properties eines Schemas

Wie zuvor bereits beschrieben, kann ein Element auch als Liste deklariert werden. Dies ist an einem kleinen Beispiel aus Listing 9 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 7 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.

Durch ein fest definiertes Schema ist es vielen IDEs, darunter auch IntelliJ und VSCo-
de, welche aus einem Schema nicht nur die fertige Datei auf Fehler überprüfen, sondern


```

{
  "title": "Products",
  "description": "A list of products from Acme's catalog",
  "type": "array",
  "items": {
    "oneOf": [
      {
        "type": "object",
        "properties": {
          "productId": {
            "description": "The unique identifier for a product",
            "type": "integer"
          }
        },
        "required": [
          "productId"
        ]
      }
    ]
  }
}

```

Listing 9: Listen Beispiel eines JSON Schemas

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 für *schemastore* unterstützen sind unter folgendem Link zu finden: <https://www.schemastore.org/json/#editors>

2.2.1.4 fulibTools

FulibTools ist der Teile Fujaba Tool Suite, auch das in dieser Arbeit erstellte fulibWorkflows ist ein Teil der Fujaba Tool Suite. Fulib bildet die Grundlage für fulibTools, wobei fulibTools erweiterte Möglichkeiten für die Nutzung von fulib bereitstellt. Fulib ist ein Codegenerator, welcher mittels einer Domain Specific Language (DSL), Modelle als Diagramme darstellen

kann.[Kas21a] Dies begrenzt sich nicht nur auf Klassen-, sondern auch auf Objektmodelle. FulibTools ist eine Erweiterung, da die Generierung der Diagramme auch abseits der eben erwähnten DSL funktioniert.[Kas21b] Hierdurch bietet sich die Möglichkeit Objektmodelle über ein spezielles YAML-Format oder ein Java Objektmodell zur Laufzeit zu generieren. Gleiches gilt für Klassenmodelle. Die Verwendung von FulibTools ist somit für diese Arbeit eine bessere Wahl, als die *Graphviz*, eine Bibliothek zur Generierung von Diagrammen, direkt zu verwenden. Dies ist der Fall, da FulibTools bereits die Arbeit der Verarbeitung eines Inputs übernimmt und hierdurch leichter für ein weiteres Tool der Fujaba Tool Suite zu verwenden ist.

2.2.2 fulibWorkflows Web-Editor Frontend

Der Web-Editor für fulibWorkflows besteht aus einem Frontend und einem Backend. Dieses Kapitel beschäftigt sich mit den Technologien, welche für das Frontend verwendet wurden. Hierbei wurde die Entscheidung über die verwendeten Technologien für die Integrierung des Editors in die Webseite <https://fulib.org/editor>. Über eine Integrierung wird in Kapitel 6 näher eingegangen.

2.2.2.1 Angular

Die Grundlage für das Frontend ist Angular. Angular ist ein Framework für das Designen von Applikationen und gleichzeitig eine Entwicklungsplattform.[Goo21] Für Entwickler ist das Angular Command-Line-Interface ein wichtiger Bestandteil bei der Entwicklung mit Angular. Neben der Generierung einer neuen Anwendung, können ebenfalls neue Komponenten, Services und Module generiert werden. Die *Komponenten* dienen der Strukturierung einer Anwendung und enthalten verschiedene Abschnitte einer Anwendung. Hierbei besteht ein großer Vorteil darin, Komponenten modular zu gestalten. Dies bedeutet, dass man Komponenten so entwickelt, dass diese in der Anwendung wiederverwendet werden können, sollte dies möglich sein. Eine Komponente besteht aus drei einzelnen Bereichen:

1. Der Logik/dem Code, welche/r in Typescript verfasst wird.
2. Einem Template, welches eine HTML-Datei ist.
3. Styles, welche im Template eingebunden werden können, um die Oberfläche grafisch zu verändern.

Im Template werden neben den Styles auch Bestandteile des Code-Segments verwendet um Daten dynamisch anzeigen zu können. Ein neu generiertes Angular Projekt bietet allerdings nur die Grundlage einer Anwendung. Um weitere Funktionalitäten in der Anwendung

verwenden zu können, können sogenannte Package Manager, wie *npm* oder *yarn* verwendet werden, um neue Bibliotheken einzubinden.

Dies soll für dieses Kapitel genügen, da Angular in Gänze zu erklären den Rahmen dieser Arbeit überschreiten würde. In Kapitel 3.2 wird auf weitere Funktionen genauer eingegangen und dies anhand der Implementierung erklärt. Folgend werden eben erwähnte Bibliotheken erläutert, welche die wichtigsten Bestandteile der Anwendung sind.

2.2.2.2 Bootstrap

Um eine ansehnlichere Oberfläche zu gestalten, welche einheitlich mit der von *fulib.org* sein soll, wurde Bootstrap zum Stylen der OberflächenElemente verwendet. Bootstrap bietet diverse Komponenten, wie Eingabefelder, Buttons, Menüs, Pop-Ups und viele weitere. Neben den Styles enthalten diese auch zusätzliche Funktionen, welche im Kontext der Komponente sinnvoll sind. Dabei bleibt es weiterhin abänderbar, um dem Entwickler mehr Freiheiten zur Gestaltung einer Oberfläche zu geben. Weiterhin ermöglicht es Bootstrap das Layout, also die Anordnung, von Komponenten auf einer Seite zu sortieren. Dies beginnt bei der Größe einer Komponente bis hin zu der Anordnung in der Horizontale und Vertikale.[Tea21b]

Zusätzlich zu Bootstrap ist es möglich Bootstrap Icons zu verwenden. Hierbei handelt es sich über 1500 Icons, welche im open source Prinzip zugänglich sind.[Tea21a]

2.2.2.3 CodeMirror

CodeMirror ist ein Texteditor, welcher in JavaScript geschrieben wurde und somit in Web-Anwendungen verwendet werden kann. Es gibt zahlreiche Optionen, um CodeMirror an die Bedingungen der zu bauenden Anwendung anzupassen. Neben zahlreichen Programmiersprachen, welche durch das Hervorheben von Schlüsselwörter und der Überprüfung der Syntax, emuliert werden können, ist es möglich CodeMirror zu einer eigenen, individualisierten IDE zu machen. Erweiterungen für einen CodeMirror sind die sogenannten Add-Ons. Hierbei gibt es neben vielen bereits vorhandenen Erweiterungen auch die Möglichkeit eigene Add-Ons zu erstellen. Hierzu zählt unter anderem das zuvor erwähnte farbliche Hervorheben von Schlüsselwörtern, auch Highlighting genannt, wie auch die Überprüfung des Codes auf die Syntax der eingestellten Programmiersprache.[Hav21]

Für eine einfache Einbindung in ein Angular-Projekt existieren bereits mehrere Bibliotheken, welche eine CodeMirror Komponente bereitstellen. Bei dieser Komponente können nicht nur Optionen übergeben, sondern auch der Inhalt eines CodeMirrors, also den ge-

schriebenen Code, aus der Komponente extrahiert werden. In dieser Arbeit wurde hierzu ngx-codemirror von Scott Cooper verwendet.[Coo21]

2.2.3 fulibWorkflows Web-Editor Backend

Das folgende Kapitel beschäftigt sich mit dem zweiten Bestandteil des Web-Editors, das dazugehörige Backend. Vom Frontend wird eine YAML-Beschreibung ans Backend geschickt, in diesem wird dies als Eingabe für fulibWorkflows verwendet. Da fulibWorkflows eine Java-Bibliothek ist, benötigt es ein Backend basierend auf Java.

2.2.3.1 Spring Boot

Mittels *Spring Boot* ist es möglich schnell und ohne zusätzliche Konfiguration eine auf *Spring* basierende Applikation zu erstellen.[VMw22a] Spring ist ein Framework, welches sich als Ziel gesetzt hat Java Programmierung zu vereinfachen und zu verschnellern, allerdings keine Einbußen bei Geschwindigkeit, Komplexität und Produktivität zu machen.[VMw22b] In diesem Kapitel wird sich mit der Erstellung eines Rest-Services eingegangen. Ein Rest-Services stellt Endpunkte bereit, welche über REST angesprochen werden können. Diese eignen sich zur Nutzung als simples Backend.

The screenshot shows the Spring Initializr web application interface. The header includes the Spring logo and 'spring initializr' text. A hamburger menu is on the left, and a settings icon is on the right. The main content area is divided into two columns. The left column contains configuration options: 'Project' (Maven Project, Gradle Project), 'Language' (Java, Kotlin, Groovy), 'Spring Boot' (2.7.0 (SNAPSHOT), 2.6.3 (SNAPSHOT), 2.6.2, 2.5.9 (SNAPSHOT), 2.5.8), 'Project Metadata' (Group: com.example, Artifact: demo, Name: demo, Description: Demo project for Spring Boot, Package name: com.example.demo), 'Packaging' (Jar, War), and 'Java' (17, 11, 8). The right column is titled 'Dependencies' and features a button 'ADD DEPENDENCIES... CTRL + B'. Below this, 'Spring Web' is selected with a 'WEB' tag, and a description reads: 'Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.'

Abbildung 2.2: Spring Initializr für eine Web Anwendung

Das Anlegen eines Spring Boot Projektes kann durch den *Spring Initializr* erledigt werden.[VMw21] In Abbildung 2.2 ist dieser zu sehen. Hierbei können diverse Einstellungen getätigt werden, um das zu generierende Projekt an die Anforderungen des Generierenden anzupassen. Die Abbildung zeigt die getätigten Einstellungen, um ein Gradle Projekt mit Java, Version 17, als Programmiersprache zu generieren. Ebenfalls kann die Version von Spring Boot eingestellt werden. Zusätzliche Dependencies können im rechten Bereich

des *Spring Initializers* hinzugefügt werden. In diesem Fall wurde **Spring Web** ausgewählt, wodurch die wichtigsten Dependencies für eine REST Applikation bereits enthalten sind.

Das aus den vorherigen Einstellungen beschriebene Projekt ist bereits ein fertiges Backend, welches direkt ausgeführt werden kann. Durch sogenannte Annotations, welche mit einem @ gekennzeichnet werden, ist es möglich weitere Endpunkte für das Backend zu definieren. Auf die genaueren Funktionen, welche mittels Annotations implementiert werden können, wird in 3.3 eingegangen, anhand von Beispielen der Implementierung.

2.2.4 Deployment

Im folgenden Kapitel werden die Technologien, welche für die Bereitstellung einer Java-Bibliothek und einer Web-Anwendung, verwendet werden können. Hierbei wird MavenCentral für Java-Bibliotheken und Heroku für Web-Anwendungen festgelegt, da diese für die in dieser Arbeit erstellten Anwendungen verwendet wurden.

2.2.4.1 MavenCentral

Maven Central ist ein Archiv von Software Artefakten für Java Entwicklung. Artefakte können Software Development Kits (SDK)s oder Bibliotheken sein, welche von Entwicklern bereitgestellt werden, um diese zentralisiert bereitzustellen und den Aufwand der Konfiguration für andere Entwickler möglichst gering zu halten.[Cla22] Veröffentlichte Artefakte können in verschiedenen Build Management Tools verwaltet werden, zum Beispiel Gradle oder Maven. *Maven Central* wurde von der Apache Software Foundation im Jahr 2002 ins Leben gerufen.

Das Bereitstellen einer Bibliothek auf Maven Central wird anhand der Implementierung in Kapitel 3 genauer anhand von `fulibWorkflows` erläutert.

2.2.4.2 Heroku

Der in dieser Arbeit erstellte Web-Editor soll für jedermann zugänglich sein. *Heroku* ist eine Möglichkeit kostenlos und schnell Web-Anwendungen bereitzustellen. Bei Heroku handelt es sich um eine sogenannte Cloud Platform as a Service (CPaaS), welche es ermöglicht in einer Cloud Umgebung Services verschiedener Arten und Programmiersprachen bereitzustellen.[Sal22b]

Um eine Anwendung über Heroku bereitzustellen bedarf es einem Account bei Heroku. Dies ist für kleine Anwendungen oder zum Ausprobieren bereits ausreichend. Entwickler können

bis zu fünf Anwendungen über einen kostenlosen Account gleichzeitig bereitstellen. Zudem erhalten Entwickler über die Heroku CLI ein weiteres Tool zum Hochladen des Codes und damit auch dem automatischen Bauen und Bereitstellen der Anwendung. Über Logs können Fehler in der Web-Ansicht einer Applikation gesichtet und analysiert werden.

Im *Heroku Dev Center* stehen zahlreiche Tutorials bereit, um verschiedenste Anwendungen bereitzustellen. Hierunter zählen unter anderem Node.js Applikationen und Java/Gradle Anwendungen.[Sal22a] Über diese Tutorials ist es möglich mit geringem Aufwand eine lokale Anwendung für das Deployment vorzubereiten. In Kapitel 3 wird genauer auf eben solche Vorbereitungen eingegangen, welche für den Web-Editor notwendig waren.

3 Implementierung

Dieses Kapitel erläutert die Implementierung der in dieser Arbeit erstellten Anwendung. Dies werden anhand der im vorherigen Kapitel erläuterten Technologien genauer erklärt. Hierbei beschäftigt sich Kapitel 3.1 mit `fulibWorkflows`, Kapitel 3.2 mit dem Frontend des `fulibWorkflows` Web-Editors und abschließend Kapitel 3.3 mit dem dazugehörigen Backend.

3.1 `fulibWorkflows`

fulib Workflows ist eine Java-Bibliothek, welche Workflow Beschreibungen als Eingabe bekommt, diese Eingabe selbst parst und daraus HTML-/FXML-Mockups, Objektdiagramme und Klassendiagramme generiert. Das Parsen wird über einen von Antlr generierten Parser übernommen, hierzu wird in Kapitel 3.1.3 genauer auf die zugrundeliegende Grammatik eingegangen. Zudem wird auf die Limitationen des Parsers und der generierten Mockups eingegangen.

3.1.1 Workflow Format

Bevor in Kapitel 3.1.3 auf den Parser und die Grammatik einer Workflowbeschreibung eingegangen wird, ist es nötig das Format in welcher ein Workflow beschrieben werden muss, näher zu beleuchten. Die Grundlage des Formats bildet YAML, hierbei handelt es sich allerdings um eine stark begrenzte YAML-Syntax, welche verwendet werden kann. Näheres zu den Beschränkungen der Syntax wird in Kapitel 3.1.2 erklärt.

In Listing10 sind alle möglichen Post-its aufgelistet. Im weiteren Verlauf wird für einen Post-It das Wort Zettel verwendet, da es sich bei jedem Element in der Beschreibung um einen einzelnen Zettel aus dem Event Storming handelt. Als Beispiel für einen Workflow ist hierbei die Registrierung eines neuen Benutzers auf einer Webseite in vereinfachter Form dargestellt. Folgend werden alle verwendbaren Zettel aus der aktuellen Version, v0.3.4, von `fulibWorkflows` aufgelistet und deren Funktion erläutert.

workflow

```
- workflow: User Registration

- user: Karli

- page:
  - pageName: Registration
  - text: Please create a new account
  - input: Username
  - input: E-Mail
  - password: Password
  - password: Repeat Password
  - button: Register

- command: Register clicked

- policy: check e-mail input

- externalSystem: E-Mail Checker

- problem: This part is always taking a lot of time somehow

- event: e-mail is valid

- service: User management

- policy: create user

- event: user data received
  username: Karli
  eMail: karli@example.com
  psw: Karli!Example42

- data: User karli
  username: Karli
  eMail: karli@example.com
  psw: Karli!Example42

- event: User created
```

Listing 10: Beispiel aller vorhandenen “Post-Its”

Zum Darstellen eines oder mehrerer Workflows benötigt es den *workflow*-Zettel. Diesem kann lediglich ein Name zugeordnet werden, allerdings werden alle weiteren Zettel unter einem workflow-Zettel eben diesem Workflow zugewiesen. Pro Workflowbeschreibung benötigt es somit mindestens einen workflow-Zettel. Im vorherigen Beispiel gibt es einen Workflow mit dem Namen *User Registration*.

service

Durch den *service*-Zettel werden Services bereitgestellt. Diese sind lediglich zur Strukturierung des Workflows und dem später daraus entstehenden Event Storming Board da. Hiermit kann erreicht werden, dass die nach einem service-Zettel folgenden Zettel von dem vorherigen Service ausgeführt werden. Im Beispiel aus Listing 10 gibt es nur einen Service, dieser kümmert sich um die Nutzerverwaltung, daher stammt auch der Name *User management*.

problem

Falls während einem Event Storming an einem bestimmten Punkt innerhalb eines Workflows Fragen bei den Beteiligten aufkommen, können diese mittels *problem*-Zettel festgehalten werden. Gleiches gilt für Probleme, welche bisher auftraten, allerdings noch nicht festgehalten wurden. Somit ist es möglich zusätzliche Informationen, welche nicht zum eigentlichen Workflow gehören, darzustellen und in der späteren Entwicklung genauer zu betrachten. Im obigen Workflow Beispiel, gibt es das Problem, dass das Validieren einer E-Mail enorm lange dauert, siehe Zeile 20.

event

Aus dem *Domain Event* aus dem Event Storming ist die verkürzte Version *event* als Indikator für einen weiteren Zettel geworden. Hierbei wird eine Beschreibung eines Events in der Vergangenheitsform als Bezeichnung verwendet. Dies bezieht sich auf eine Aktion, welche innerhalb eines Workflows, aufgetreten ist. Wie in Listing 10 in Zeile 22 und ab Zeile 28 zu sehen, können event-Zettel allein stehen oder mit weiteren Informationen angereichert werden. Weitere Informationen, die einem Event zugeordnet sind, repräsentieren Daten, welche zu der ausgeführten Aktion gehören.

user

Ein *user*-Zettel ist sehr ähnlich zu einem *service*-Zettel. Einem User wird ein Name zugeordnet. Dieser Zettel dient ebenfalls der Strukturierung eines Workflows und leitet einen Abschnitt ein, welcher aktiv von einem Nutzer durchgeführt werden muss. Im Beispielworkflow existiert ein User, welcher den Namen Karli trägt, welches in Zeile 3 zu sehen ist.

policy Eine *policy* ist wie auch beim *event* eine Abstrahierung eines Begriffes aus dem Event Storming. Ein *policy*-Zettel beschreibt somit einen Automatismus, welcher aufgrund eines vorherigen Events einen bestimmten Ablauf an Schritten ausführt. Im Beispiel aus Listing 10 reagiert die policy aus Zeile 16 auf den vorherigen Command, um automatisch zu prüfen, ob die eingegebene E-Mail valide ist.

command

Ein *command*-Zettel stellt die Interaktion eines Nutzers mit einer Webseite oder Applikation dar. Im Falle des obigen Beispiels wird in Zeile 14 der Klick auf den Knopf *Register* aus der darüber aufgeführten Page simuliert. Dies ist anhand der kurzen Beschreibung des Commands zu entnehmen.

externalSystem

Wie auch bei dem *service* und *user* dient der *externalSystem*-Zettel dazu einen Workflow zu strukturieren. In diesem Fall bedeutet dies, dass Aktionen oder Daten von einem System ausgeführt/gesendet werden, welche nicht Teil der zu entwickelnden Software gehört, für welche das aktuelle Event Storming durchgeführt wird. In Zeile 18 des Beispiels existiert ein *externalSystem*, welches für die Validierung einer E-Mail-Adresse zuständig ist. Es kann sich dabei um ein System handeln, welches von einer anderen Firma oder einem separaten Team entwickelt wird.

data

Wie in Kapitel 2.1 bereits erwähnt, ist *data* eine Erweiterung des Event Stormings. Ein *data*-Zettel benötigt als Bezeichner eine besondere Form, diese besteht zuerst aus einer Klassenbezeichnung und anschließend einem Namen für das Objekt. Dies ist nötig um im späteren Verlauf das Aufbauen eines Datenmodells zu gewährleisten und Klassen-/Objektdiagramme generieren zu können. Wie auch bei dem *event*-Zettel kann auch einem *data*-Zettel zusätzliche Informationen übergeben werden. Auf welche Art und Weise diese aufgebaut sein müssen und welche Funktionen diese innehaben, darauf wird in Kapitel 3.1.4.3 genauer eingegangen. In Zeile 33 unseres Beispiels wird ein *User* angelegt, mit den zuvor vom Event vorhandenen zusätzlichen Informationen zu Username, E-Mail und Passwort.

page

Der *page*-Zettel ist ebenfalls eine Erweiterung zum klassischen Event Storming. Aus den *page*-Zetteln eines Workflows werden später Mockups generiert, wie dies genau funktioniert wird in Kapitel 3.1.4.2 erläutert. An diesem Punkt ist es lediglich wichtig, die Restriktionen einer *page* zu erläutern. Eine Page besteht aus einer Liste an Elementen, hierbei steht *pageName* lediglich für den Bezeichner der Page und wird im Mockup nicht dargestellt. Um die Oberfläche einer Applikation zu beschreiben, existieren vier Elemente, welche beliebig oft verwendet werden können.

Ein Text Element kann enthält einen Text, welcher entweder zur Verwendung als Überschrift, Bezeichner oder Trenner verwendet werden kann. Um Daten in einer Applikation eingeben zu können, gibt es das *input*- und das *password*-Element. Bei beiden handelt es sich um Eingabefelder, mit dem Unterschied, dass bei dem *password*-Element der einge-

tippte Inhalt mit Punkten ersetzt wird. Der Bezeichner, welcher einem Input oder Password hinterlegt wird, wird als Platzhalter im Eingabefeld und als Label über eben diesem angezeigt. Zuletzt kann ein Knopf mittels dem *button*-Element erstellt werden, dabei wird der Bezeichner als Inhalt des Knopfes dargestellt.

Eine Oberfläche kann nur von oben nach unten beschrieben werden, eine weitere Limitierung ist, dass es nicht möglich ist mehrere Elemente in eine Reihe zu ordnen. Hierdurch ist das Designen eines Mockups durch die begrenzten Elemente einer Page sehr stark begrenzt.

3.1.2 JSON-Schema

Wie im vorherigen Kapitel bereits erwähnt bedient sich die Beschreibung eines Workflows grundlegend der Syntax von YAML. In Kapitel 2.2.1.3 wurden bereits die Grundlagen für JSON-Schemas geschaffen, in diesem Kapitel wird das erstellte *fulibWorkflows*-Schema genauer betrachtet. Das Schema ist komplett im Anhang hinterlegt, da dieses zu lang ist um es übersichtlich in diesem Kapitel zu erläutern. Dadurch wird nur auf die wichtigsten Punkte der Implementierung eingegangen. Um die Lesbarkeit für Entwickler zu verbessern, ist das Schema in zwei Dateien aufgeteilt.

Listing 11 ist eine minimale Version des eigentlichen Schemas, in welchem dennoch die wichtigsten Funktionen dargestellt sind. Das *fulibWorkflows*-Schema ist in zwei Hauptteile unterteilbar, in die Definitionen und die Festlegung der erlaubten Elemente in der obersten Liste. Durch das Schema werden nur JSON-/YAML-Dateien akzeptiert, welche aus einer Liste an Elementen bestehen. Hierbei sind die Elemente jedoch festgelegt durch die in Zeile 20 und 21 dargestellten Zeilen. Ein *item* darf nur aus einem der Elemente bestehen, welche in der Auflistung ab Zeile 22 festgelegt sind. Um die Lesbarkeit zu vereinfachen und die Schachtelungstiefe möglichst gering zu halten, werden die erlaubten Elemente lediglich referenziert. Die referenzierten Elemente wurden im ersten Teil des Schemas, ab Zeile 4, definiert. Für jedes der im vorherigen Kapitel erwähnten Zettel, gibt es ein Element im Schema. Die Definitionen der Elemente, welche nicht in Listing 11 dargestellt sind, enthalten lediglich Standardwerte, welche bereits in Kapitel 2.2.1.3 erläutert wurden. Diese können im Anhang nachgelesen werden. Die Definition der Page ist jedoch ein Sonderfall, welche durch das Aufteilen in mehrere Dateien entstanden ist. Es ist möglich weitere Schemas aus einer anderen Datei zu importieren, dies ist in Zeile 11 ersichtlich. Auch dort wird erneut eine Referenzierung durchgeführt, allerdings nicht auf eine im Schema befindlichen Definition, sondern auf die Page Definition aus der *page.schema.json* Datei. Die Aufteilung wurde durchgeführt, da die Page eine Liste ist und ebenfalls fünf eigene Elemente definiert. Allein das Page-Schema beläuft sich auch 93 Zeilen und umfasst somit allein die Hälfte des Parent-Schemas.

```
1  {
2    "type": "array",
3    "additionalItems": false,
4    "$defs": {
5      ....
6      "pageItem": {
7        "description": "Defines a ui page",
8        "type": "object",
9        "properties": {
10         "page": {
11           "$ref": "page.schema.json"
12         }
13       },
14       "required": [
15         "page"
16       ],
17       "additionalProperties": false
18     }
19   },
20   "items": {
21     "oneOf": [
22       ....
23       {
24         "$ref": "#/$defs/pageItem"
25       },
26       {
27         "$ref": "#/$defs/problemItem"
28       }
29     ]
30   }
31 }
```

Listing 11: Referenzieren eines anderen Schemas

Wie in Kapitel 2.2.1.3 bereits erwähnt, ist das `fulibWorkflows`-Schema ebenfalls bei `Schemastore.org` hinterlegt. Es wird automatisch für Dateien mit der Dateierweiterung `.es.yaml` vom Editor verwendet. Dadurch ist es zum Beispiel in IntelliJ möglich Autovervollständigung für `fulibWorkflows` zu bekommen und auf Fehler hingewiesen zu werden.

In Abbildung 3.1 ist das Hervorheben von Fehlern in IntelliJ dargestellt, welches durch das JSON-Schema generiert wird. In Abbildung 3.1(a) ist die Datei leer, wodurch die Schema Validierung anspricht und den Entwickelnden darauf hinweist, dass ein Array, also eine Liste an Elementen, benötigt wird. Sobald ein Element begonnen wird, wird diese Warnung nicht mehr angezeigt. Sollte der Entwickelnde wiederum ein Element hinzufügen, welches keinem der definierten Elemente des Schemas entspricht, wird die Meldung aus Abbildung 3.1(b) angezeigt. Da ein Command keine zusätzlichen Attribute/Properties akzeptiert, dennoch eines hinzugefügt wurde, besagt der Fehler, dass es nicht erlaubt ist weitere Attribute hinzuzufügen.



((a)) Leere Datei

((b)) Fehlerhafte Eingabe

Abbildung 3.1: Fehleranzeige in IntelliJ



((a)) Alle Elemente

((b)) Page Elemente

Abbildung 3.2: Autovervollständigung in IntelliJ

Wie zuvor erwähnt ermöglicht ein Schema jedoch nicht nur das Hervorheben von Fehlern, sondern unterstützt den Entwickelnden zusätzlich durch Autovervollständigung. Dies ist in Abbildung 3.2 dargestellt. Hierbei wird aufgrund des Kontextes verschiedene Möglichkeiten von zu erstellenden Elementen vorgeschlagen. Auf oberster Ebene werden alle erlaubten Schlüsselwörter für Elemente angezeigt, welches in Abbildung 3.2((a)) dargestellt ist. Hierbei fällt auf, dass die Elemente für eine Page nicht angezeigt werden, dies ist jedoch der Fall, sobald der Kontext dies zulässt. Während der Entwickelnde ein page-Element definiert, werden wie in Abbildung 3.2((b)) dargestellt die Schlüsselwörter für Page-Elemente vorgeschlagen.

3.1.3 Antlr Grammatik

Da die möglichen Eingaben durch das zuvor beschriebene JSON-Schemas bereits verringert wurden, viel die Wahl der Verarbeitung der YAML-Eingabe auf eine durch Antlr generierten Parser. Der generierte Parser bietet die Möglichkeit, während dem parsen weitere Aktionen durchzuführen und in dem Fall dieser Anwendung ein Datenmodell aus der Eingabe zu erstellen. Das Datenmodell wird im folgenden weiterverarbeitet und bietet somit eine Grundlage für die Generierung, auf welche im folgenden Kapitel eingegangen wird.

Wie in Kapitel 2.2.1.1 bereits beschrieben, ist die Grundlage eines Antlr Parser die dazugehörige Grammatik, welche nun beleuchtet wird. Die komplette Grammatik ist im Anhang hinterlegt, in diesem Kapitel werden lediglich Ausschnitte daraus verwendet.

```

5  file: workflows+ ;
6
7  workflows: workflow NEWLINE eventNote* ;
8
9  eventNote: ( normalNote | extendedNote | page) NEWLINE? ;

```

Listing 12: Grammatik für Workflows

Zuerst wird die grundlegende Struktur einer Datei festgelegt, dies ist durch die drei Regeln in Listing 12 dargestellt. Da eine Datei mehrere Workflows beinhalten kann, ist die oberste Regel in Zeile 5 der Startpunkt des Parser. Da als Eingabe eine YAML-Datei ist, heißt die oberste Regel *file* und erfordert mindestens einen workflow. Ein workflow besteht immer aus einem workflow-Note und beliebig vielen event-Notes, wobei diese immer mit einer Leerzeile von einander getrennt sind. Die Spezifikation ist aufgrund der YAML-Syntax notwendig. Ein event-Note kann einer aus drei Typen sein, wobei nach einem Note beliebig viele Leerzeilen folgen können.

```

11 workflow: MINUS 'workflow' COLON NAME ;
12
13 normalNote: MINUS NORMALNOTEKEY COLON NAME ;
14
15 extendedNote: MINUS EXTENDEDNOTEKEY COLON NAME NEWLINE attribute* ;
16
17 page: MINUS 'page' LISTCOLON NEWLINE pageList ;

```

Listing 13: Grammatik für Notes

Die Unterscheidung zwischen normal-/extended-Note, workflow und page erfolgt durch das Schlüsselwort, welches zwischen Bindestrich(MINUS) und Doppelpunkt(NAME) befindet. Sowohl ein workflow-Note als auch die normal-Notes besitzen lediglich nach dem Doppelpunkt einen Wert, welcher durch **NAME** gekennzeichnet ist. Dies ist in Listing 13 in Zeile 11 und 13 dargestellt. Ein extended-Note besitzt neben dem Wert zusätzliche Attribute, welche in einer neuen Zeile beschrieben werden. Die Anzahl der attribute ist beliebig, es ist somit erlaubt einen extended-Note ohne weitere Attribute anzugeben. Die Page ist wie zuvor bereits häufiger erwähnt ein Sonderfall, welches sich auch in der Grammatik widerspiegelt. Dem Schlüsselwort *page* folgt ein gesonderter Doppelpunkt und anschließend eine Liste von neuen Elemente.

```

30 NORMALNOTEKEY: 'externalSystem' | 'service' | 'command'
31               | 'policy' | 'user' | 'problem' ;
32
33 EXTENDEDNOTEKEY: 'event' | 'data' ;

```

Listing 14: Schlüsselwörter zum Identifizieren von Notes

Die zuvor erwähnte *normalen Notes* bestehen wie in Listing 14 Zeile 30 und 31 dargestellt aus externalSystem, service, command, policy, user und problem. Diese erhalten lediglich

einen Bezeichner und erlauben keine weiteren Attribute. Zu den *extended Notes* zählen lediglich event und data. Diese erhalten wie zuvor beschrieben weitere Attribute um Daten, welche zwischen Services verschickt werden, darstellen zu können.

```
19 attribute: INDENTATION NAME COLON value NEWLINE? ;
20
21 value: NAME | NUMBER | LIST;
```

Listing 15: Grammatik von Attributen

Attribute werden eingerückt und enthalten neben einem Bezeichner(NAME) einen dazugehörigen Wert(value). Ein Wert kann entweder ein Text, eine Nummer oder eine Liste sein, wobei eine neue Zeile optional ist. Die dazugehörigen Regeln sind in Zeile 19 und 21 aus Listing 15 vermerkt.

Wie in Listing 16 genauer beschrieben, ist der akzeptierte Text auf eine feste Zahl an verschiedenen Zeichen begrenzt. Ein Text muss stets mit einem Buchstaben beginnen, ungeachtet ob groß oder klein geschrieben. Darauf können Zahlen, Sonderzeichen und weitere Wörter folgen. Die Sonderzeichen sind in Zeile 36 genauer beschrieben.

```
36 NAME: ([A-Za-zäöüÿß] [0-9]* [-/_,. ' @! ?]* [ ]* [0-9]* [-/_,. ' @! ?]* [ ]*)+ ;
37
38 LIST: '[' (.) *? ']' ;
39
40 NUMBER: [0-9]+ ;
```

Listing 16: Grammatik von Werten

Eine Nummer kann lediglich eine ganze Zahl sein, führende Nullen sind erlaubt. Die Möglichkeit als Wert eine Liste angeben zu können, basiert auf der Möglichkeit Objekt- und Klassendiagramme mit `fulibWorkflows` generieren zu können. Hierzu wurde die Syntax von Java als Grundlage genommen. Zwischen den Klammern in Zeile 38 befindet sich eine sogenannte Wildcard, welche es erlaubt alle Symbole als Eingabe zu akzeptieren. Die Klammern erfüllen somit nicht nur den Zweck als Listendarstellung, sondern auch die Begrenzung der Wildcard. Eine Wildcard für die Eingabe eines Textes zu verwenden war für diese Grammatik aufgrund der Struktur vorerst nicht möglich, da es keine passenden Begrenzungen gab, welche keine anderen Regeln überschrieben hätte.

```
23 pageList: pageName NEWLINE element* ;
24
25 pageName: INDENTATION MINUS 'pageName' COLON NAME ;
26
27 element: INDENTATION MINUS ELEMENTKEY COLON NAME NEWLINE ;
```

Listing 17: Grammatik einer Page

Jeder Page muss ein `pageName` Element zugeordnet werden, um diese später referenzieren zu können. Weiterhin können Pages beliebig viele Elemente beherbergen. Ein Element wiederum muss entweder als Bezeichner `text`, `input`, `password` oder `button` besitzen, um valide zu sein. Jedem dieser Elemente wird anschließend ein `Text(NAME)` zugeordnet, welche Funktion diese für das jeweilige Element erfüllen, wurde bereits in Kapitel 3.1.1 erläutert.

Wie zuvor bereits beschrieben, wird aus der Grammatik ein Parser generiert. Dazu gehört unter anderem ein Interface, welches für diese Grammatik den Namen *FulibWorkflowsListener* trägt. Um während dem Parsen ein Datenmodell aufzubauen, wurde eine eigener Listener implementiert, welcher die Methoden des Interfaces überschreibt. Für jeder Regel aus der Grammatik existiert eine `enter`- und eine `exit`-Methode. In den `enter`-Methoden werden lediglich neue Objekte angelegt und globale Variablen zurückgesetzt.



Abbildung 3.3: Klassendiagramm fulibWorkflows

Bevor anhand eines Beispiels die Verwendung einer `exit`-Methode erläutert wird, ist es notwendig das Datenmodell genauer zu betrachten. In Abbildung 3.3 ist das Klassendiagramm abgebildet, welches die Struktur eines Event Storming Boards nach dem Parsen der YAML-Eingabe widerspiegelt. Jeder Note besitzt eine dazugehörige Klasse, welche von `BaseNote` erbt. Für jeden Note existiert somit ein Name und ein Index, auf welchen im folgenden Abschnitt eingegangen wird. Da es im Event Storming ein dazugehöriges *Board* gibt, ist dies ebenfalls eine Klasse, welche alle *workflows* einer Eingabe hält. Ein Workflow besteht weiterhin aus vielen Notes. Wie zuvor bereits erläutert, sind `Data`, `Event` und `Page` Sonderfälle unter den Notes, da diese weitere Daten beherbergen. Daher haben diese Klassen ein Attribut, welches diese Daten organisiert in einer Map hält. Hierbei wird als

key der Index eines Notes verwendet und das value ist ein Pair. Das Pair beinhaltet den Bezeichner und den dazugehörigen Wert einer zusätzlichen Property eines Notes.

```
106  @Override
107  public void exitPage(FulibWorkflowsParser.PageContext ctx){
108      Page newPage = new Page();
109
110      newPage.setContent(noteData);
111      newPage.setIndex(noteIndex);
112      noteIndex++;
113
114      notes.add(newPage);
115  }
```

Listing 18: exitPage-Methode

In Listing 18 wird ein neues Page-Objekt erstellt. Bevor jedoch die exitPage-Methode aufgerufen wird, werden alle Elemente in der Variable *noteData* gespeichert. Dieser werden in der exitElement-Methode hinzugefügt, der Name einer Page wird hingegen in der gesonderten exitPageName-Methode zu *noteData* hinzugefügt. Zusätzlich zu den Daten eines Notes, wird diesem ein Index gegeben und anschließend zur Liste aller Notes hinzugefügt. Der Index ist wichtig, um die Reihenfolge von Notes und deren Attributen beizubehalten.

3.1.4 Generierung von Dateien

Aus einer workflow beschreibung können bis zu fünf verschiedene Typen von Dateien generiert werden. Der Einstiegspunkt und somit der Start des Parsens und der Generierung ist die *BoardGenerator*-Klasse. Diese hat Methoden um Eingaben entweder von einer Datei oder eines Strings weiterzuverarbeiten. Nachdem mittels des Parsers aus der Eingabe ein fertiges Board-Objekt erstellt wurde, werden weitere Klassen zur Generierung verwendet. HTML-Dateien werden vom *HtmlGenerator* generiert, hierbei handelt es sich um Mockups und das Event Storming Board. Mockups können allerdings auch als FXML-Datei generiert werden, um eine Grundlage für eine JavaFx-Anwendung zu bilden, diese Generierung übernimmt der *FxmlGenerator*. Zuletzt vereint der *DiagramGenerator* die Generierung von Objekt- und Klassendiagrammen. Außer der *BoardGenerator*-Klasse sind die restlichen Generator-Klassen für die Vorbereitung der Daten zuständig. Diese erhalten Eingaben von dem *BoardGenerator*, bereiten diese Eingabe auf, je nachdem welche Daten benötigt werden und enthalten eine separate Methode zum Erstellen von Dateien im Dateisystem. Das Bauen einer Datei in Form eines Strings wird in einer gesonderten *Constructor*-Klasse erledigt. Da es fünf verschiedene Typen von Dateien gibt und das Bauen für jede Datei anders ist, existieren fünf *Constructor*-Klassen, für jeden Dateityp eine. Im Folgenden werden die Aufbereitungsschritte genauer beleuchtet.

3.1.4.1 Event Storming Board

Für die Generierung des Event Storming Boards bedarf es keiner Bearbeitung des *Html-Generators*, da das gesamte Board generiert werden soll und die Daten, welche vom Parser erstellt wurden bereits optimiert sind. Die HTML-Datei wird mittels *StringTemplates*, welche in einer *StringTemplateGroup* organisiert sind, zusammengebaut. Für jeden workflow im Board-Objekt wird eine neue Reihe in der HTML-Datei angelegt. Innerhalb einer workflow-Reihe werden alle dazugehörigen Notes gebaut. Hierbei wird zwischen den verschiedenen Notes unterschieden, um verschiedene Darstellungen zu ermöglichen. Je nach Note wird eines von drei *StringTemplates* verwendet. Für die Standard-Notes wird lediglich eine neue *Card*, eine Bootstrap CSS-Klasse, erstellt, welche den Typen des Notes, dessen Content und eine bestimmte Farbe übergeben bekommt. Für die organisatorischen Notes, User, Service und ExternalSystem wird eine Card erstellt, welche kleiner als die eines normalen Notes ist. Zudem wird in organisatorischen Notes nur ein Icon und der Bezeichner angezeigt, wobei das Icon ein Bootstrap Icon ist. Data- und Page-Notes werden gesondert mit einem dritten *StringTemplate* behandelt, da es neben Typ, Content und Farbe noch einen Link gibt, welcher als Button definiert und für die Verwendung im WebEditor genutzt wird.



Abbildung 3.4: Mittels fulibWorkflows generiertes Event Storming Board

In Abbildung 3.4 ist ein Ausschnitt des in Listing 10 beschriebenen Workflows in Form eines generierten Event Storming Boards dargestellt. Hierbei sind die zuvor beschriebenen Unterschiede zwischen den verschiedenen Notes erkennbar. Jeder Note-Typ besitzt eine eigene Farbe, wobei für User, Service und ExternalSystem eine kleinere Card und jeweils ein eigenes Icon erkennbar sind. Ebenfalls ist der Page-Note der Note mit den meisten Informationen in diesem Ausschnitt, da nur dort zusätzliche Informationen in der Workflowbeschreibung existierten.

3.1.4.2 Mockups HTML/FXML

Die Generierung von HTML-Mockups wird durch die zuvor bereits erwähnte *HtmlGenerator*-Klasse übernommen. Diese filtert aus allen Workflows und den dazugehörigen Notes die Pages heraus und übergibt diese an die *PageConstructor*-Klasse. FXML-Mockups erhalten eine gesonderte Generator- und Constructor-Klasse. Die in diesen Klassen befindlichen Funktionen ähneln der Funktionsweise des *HtmlGenerators* und des *PageKonstruktors* stark. Eine Unterscheidung in HTML und FXML wurde lediglich vorgenommen, um bei der

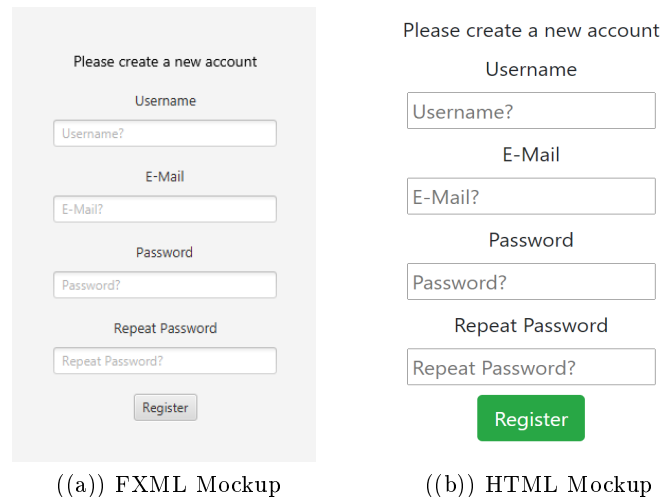


Abbildung 3.5: Mittels fulibWorkflows generierte Mockups

Generierung die Möglichkeit von verschiedenen Option offenzulassen. Somit könnten nur HTML- oder FXML-Mockups erstellt werden. Der FxmlConstructor und PageConstructor haben somit sowohl eine ähnliche Funktionsweise als auch einen ähnlichen Aufbau, da die zugrundeliegenden Daten die gleichen sind. Lediglich die zugrunde liegende String Template Group (STG) unterscheidet die beiden Klassen. In beiden STGs gibt es ein String Template (ST) zum Aufbau der generellen Struktur der jeweiligen Datei und für jedes verfügbare Oberflächenelement ein weiteres ST. Es existieren somit für die Oberflächenelemente STs für Text, Eingabefeld, Passwortfeld und Knopf.

In Abbildung 3.5 sind die generierten Mockups aus dem vorherigen Beispiel dargestellt. Diese bestehen, entsprechend der Beschreibung, aus einem Text, drei Eingabefeldern, einem Passwortfeld und einem Knopf. Bei der Generierung wird vor allem darauf geachtet, dass sich die Oberflächen möglichst stark ähneln. Der Aufbau der in Abbildung 3.5((a)) und Abbildung 3.5((b)) Oberflächen ist somit identisch, Unterschiede sind lediglich durch das Stylen des HTML-Mockups, mittels Bootstrap, zu erkennen.

3.1.4.3 Objektdiagramme

Zuletzt gibt es die *DiagramGenerator*-Klasse, welche die Eingabe für Objekt- und Klassendiagramme aufarbeitet. Jeder Data-Note erhält sein eigenes Objektdiagramm. Damit die zeitliche Abfolge und somit ein korrektes Objektdiagramm entsteht, wird jeder Data Note zu einer Liste hinzugefügt und diese anschließend an die *ObjectDiagramConstructor*-Klasse übergeben. Hierdurch werden pro Data Note mehr Objekte zu dem dazugehörigen Objektdiagramm hinzugefügt, sollte es eine Verbindung zu einem bestehenden Objekt geben. FulibTools verwendet Graphviz zur Erstellung von Diagrammen, zusätzlich besitzt FulibTools die Möglichkeit Objekt- und Klassendiagramme anhand einer bestimmten Eingabe zu generieren. Diese Funktionalität macht sich fulibWorkflows im ObjectDiagramConstructor

zunutze. Aus der übergebenen Liste an Data Notes baut der `ObjectDiagramConstructor` eine YAML-Datei, welche den Spezifikationen von `fulibYaml` entspricht. In Listing 19 ist die zum `ObjectDiagramConstructor` gehörende STG-Datei abgebildet. Ein Objekt benötigt immer einen Namen, eine Klasse in Zeile 5 als *type* gekennzeichnet und optionale Attribute. Die Form eines Attributes ist in dem ST ab Zeile 10 abgebildet, ein Attribute besteht lediglich aus einer Klasse, erneut *type* genannt, und dem dazugehörigen Wert.

```

1  delimiters "{", "}"
2
3  object(name, type, attributes) ::= <<
4  - {name}: .Map
5    type: {type}
6    {attributes}
7
8  >>
9
10 attribute(type, value) ::= <<
11 {type}: {value}
12
13 >>

```

Listing 19: `FulibYaml.stg`

Nachdem ein Objektdiagramm in `fulibYaml` Notation vorhanden ist, wird `FulibTools` verwendet um daraus ein Diagramm zu generieren. In Listing 20 ist diese Generierungsmethode abgebildet. Der erste Parameter der Methode enthält die Objektstruktur in Form von `fulibYaml` als `String`. Daraus wird in Zeile 99 und 100 ein `root`-Objekt erstellt, welches die Klasse `YamlIdMap` aus der Bibliothek `fulibYaml` nutzt. Dieses `root`-Objekt wird gemeinsam mit dem in Zeile 96 festgelegten Dateinamen durch `FulibTools` in Zeile 102 generiert. `FulibTools` erlaubt bei der Generierung nicht, dass der Inhalt der zu generierenden Datei zurückgegeben wird, die Datei wird sofort im Dateisystem erstellt.

Im Anschluss an die Generierung durch `FulibTools` wird der Inhalt der generierten Datei als `String` ausgelesen und von der Methode zurückgegeben. Danach wird die generierte Datei, sowie ein Ordner wieder gelöscht. Dies hat den Hintergrund, dass die Generierung von Dateien, in Form vom Speichern im lokalen Dateisystem, gesammelt in der `BoardGenerator`-Klasse vollzogen werden soll. Zudem wird im `BoardGenerator` die Möglichkeit geboten, Dateien aus einer Workflow Beschreibung zu generieren und diese als `String` zurückzugeben. Diese Methoden existieren für die Nutzung im `fulibWorkflows` Web-Editor, später dazu mehr.

In Abbildung 3.6 sind zwei Objektdiagrammen dargestellt, welche mit `fulibWorkflows` generiert wurden. Die Beschreibung des workflows stammt von einem Beispiel, welches zum Testen dieser Funktionalität in `fulibWorkflows` verwendet wurde.[Kün21] Zu dem Objektdiagramm aus Abbildung 3.6((a)) ist ein weiteres Room-Objekt in Abbildung 3.6((b)) hinzugefügt worden. Dies basiert auf der vorher erwähnten Funktion, dass Objekte in einer

```

95 private String generateObjectDiagram(String objectYaml, int index) {
96     String fileName = "tmp/test/diagram_" + index;
97     String result = "";
98
99     YamlIdMap idMap = new YamlIdMap();
100    Object root = idMap.decode(objectYaml);
101
102    fileName = FulibTools.objectDiagrams().dumpSVG(fileName, root);
103
104    try {
105        result = Files.readString(Path.of(fileName + ".svg"));
106
107        Files.deleteIfExists(Path.of(fileName + ".svg"));
108
109        Files.deleteIfExists(Path.of("tmp/test/"));
110    } catch (IOException e) {
111        e.printStackTrace();
112    }
113
114    return result;
115 }

```

Listing 20: Generierungsmethode eines Objektdiagramms

Liste gespeichert werden und jedes neue Objekt dort hinzugefügt wird, bevor ein Objektdiagramm generiert wird.

3.1.4.4 Klassendiagramme

Der in der vorherigen Sektion erwähnte DiagramGenerator verwaltet nicht nur die Objektdiagramme. Nachdem alle Data Notes aus allen Workflows durchlaufen wurden, enthält die ebenfalls zuvor erwähnte Liste alle Data Notes eines Event Storming Boards. Diese Liste wird an die *ClassDiagramConstructor*-Klasse weitergegeben.

Für das Erstellen eines Klassenmodells aus Data Notes müssen die Data Notes bestimmte Spezifikationen erfüllen, um das gewünschte Ergebnis zu erzielen.

```

5 - data: Student carli
6   motivation: 83
7   stops: [stop1]
8   stops.back: student
9
10 - data: Stop stop1

```

Listing 21: Beispiel eines richtigen Data Notes

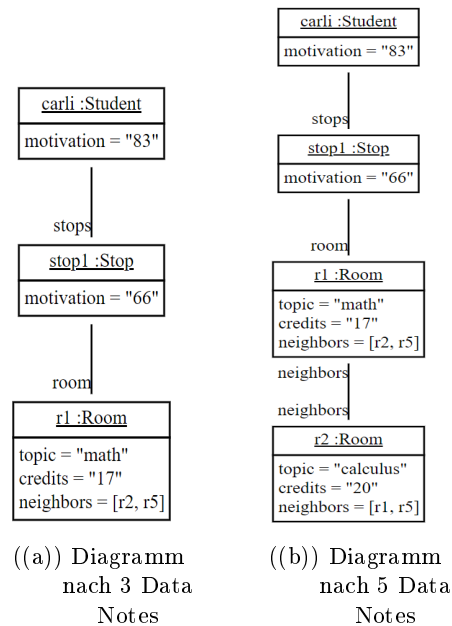


Abbildung 3.6: Mittels fulibWorkflows generierte Objektdiagramme

Anhand des Data Notes aus Listing 21 werden folgend die eben erwähnte Spezifikationen näher betrachtet. Der *Name*, welcher in Zeile 5 nach dem Doppelpunkt vergeben wird muss immer die Form: <Klassenname> <Objektname> besitzen. Zudem muss beim ersten Aufkommen einer Assoziation sowohl die Hin- als auch Rückrichtung beschrieben werden. Eine Assoziation wurde in Zeile 7 und 8 beschrieben, bei dieser wird mittels *stops* der Bezeichner für die Hinrichtung festgelegt. Mit den eckigen Klammern um den Text *stop1* wird ausgesagt, dass die Hinrichtung eine To Many-Assoziation ist, somit ein Student mehrere Stops haben kann. Die Rückrichtung wird in Zeile 8 durch *stops.back* definiert, gemeinsam mit dem Wert *student* wird der Bezeichner der Rückrichtung auf *student* gesetzt. Durch die fehlenden eckigen Klammern ist die Kardinalität der Rückrichtung 0 oder 1.

```

39  ....
40  ClassModelManager mm = new ClassModelManager();
41
42  // Create a map String,Clazz containing every possible class from the Data notes
43  createClazz(mm);
44
45  // Build all associations and put it into a global list
46  // Also Build a list of attributes, that are not allowed to be created
47  buildAssociations();
48
49  // Create all attributes
50  createAttributes(mm);
51
52  // Create all associations
53  createAssociations(mm);
54  ....

```

Listing 22: Schritte zum Aufbau eines Klassenmodells

Anhand dieser Anforderungen an ein Data Note ist es möglich ein Klassenmodell aus den Notes zu abstrahieren. In Listing 22 sind die Schritte, welche zum Aufbau eines Klassenmodells nötig sind aufgelistet. Diese Stammen samt Kommentaren aus der Implementierung, die *ClassModelManager*-Klasse ist in der fulib-Bibliothek enthalten. Im ersten Schritt, Zeile 43, werden alle Klassen in einer Map abgelegt, wobei als Key der Name der Klasse und als Value ein *Clazz*-Objekt fungiert. Hierbei wird über alle Data Notes iteriert und die darin enthaltenen Klassen aus dem Namen des Data Notes extrahiert. Schritt Zwei, Zeile 47, werden die vorhandenen Assoziationen zusammengebaut. Hierfür wurde eine Klasse namens *Association* erstellt, welche zum Zwischenspeichern der für eine Assoziation wichtigsten Daten trägt. Darunter zählen Klasse, Bezeichner und Kardinalität für Hin- und Rückrichtung. Neben diesen Daten wird in einer Liste von Strings Schlüsselwörter gespeichert, welche als Assoziation fungieren und nicht als Attribut betrachtet werden dürfen. Die wichtigsten Spezifikationen für eine funktionierende Assoziation wurde bereits zuvor anhand von Listing 21 erläutert. Weiterhin ist es wichtig, dass die verwendeten Objektnamen in der Workflowbeschreibung konsistent sind, da ansonsten keine Zielklasse ermittelt werden kann. Während dem Bauen einer Assoziation wird über alle Data Notes iteriert um die Informationen der Zielklasse zu erhalten. Der nächste Schritt, Zeile 50, erstellt Attribute für alle Klassen. Hierbei kommt die zuvor erstellte Liste an Schlüsselwörtern zum Tragen, um auszuschließen, dass eine Assoziation ebenfalls als Attribut aufgefasst wird. Im letzten Schritt zur Abstrahierung eines Klassenmodells aus den Data Notes werden die zuvor in Zeile 47 gebauten Assoziationen im Klassenmodell erstellt. In diesem Schritt wird darauf geachtet, dass alle benötigten Informationen für eine Assoziation vorhanden sind. Durch die Nutzung von fulib und dem ClassModelManager muss nicht auf Duplikate bei Attributen oder Assoziationen geachtet werden.

Nachdem das Klassenmodell erstellt wurde, wird dieses ebenfalls mit FulibTools zu einem Diagramm weiterverarbeitet. Der Ablauf hierbei ist ähnlich zu der Generierung von Objektdiagrammen. Nachdem das Diagramm generiert wurde, wird dieses ebenfalls auf dem Dateisystem gelöscht und der Inhalt des Diagramms als String zurückgegeben.

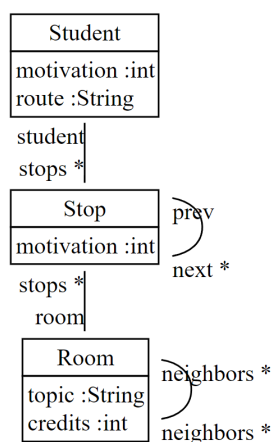


Abbildung 3.7: Mittels fulibWorkflows generiertes Klassendiagramm

In Abbildung 3.7 ist das aus der vorherigen Sektion verwendete Beispiel zur Grundlage der Generierung eines Klassendiagramms genutzt worden. Das Klassendiagramm besteht aus drei Klassen und beinhaltet alle beschriebenen Assoziationen bestehend aus Kanten, Bezeichnern an beiden Seiten und den entsprechenden Kardinalitäten. Bei den Kardinalitäten kennzeichnet ein * eine *TO MANY*-Assoziation. Die Typen der Felder einer Klasse können lediglich zwischen String und int unterschieden werden.

3.2 fulibWorkflows Web-Editor Frontend

Nachdem die erste Hälfte der Implementierung durch fulibWorkflows abgeschlossen ist, konzentrieren sich dieses und das folgende Kapitel um den dazugehörigen Web-Editor. Der Web-Editor besteht aus einem Frontend und einem Backend, welche beide über Heroku deployed wurden und somit erreichbar sind. Erreichbar ist der Web-Editor unter <https://workflows-editor-frontend.herokuapp.com/>.

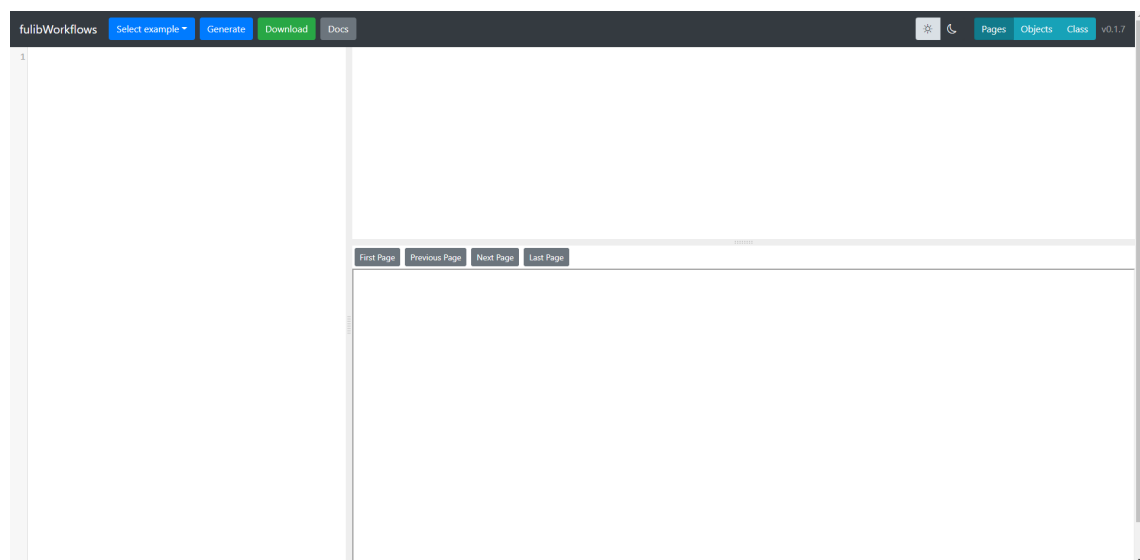


Abbildung 3.8: FulibWorkflows Web-Editor Oberfläche

In Abbildung 3.8 ist die Oberfläche des Web-Editors dargestellt. Dieser besteht aus vier verschiedenen Bereichen, welche jeweils andere Funktionen bereitstellen. Der erste dieser Bereiche ist die Navigationsleiste, welche den oberen Rand der Oberfläche einnimmt. In dieser steht zuerst, von links nach rechts, ein Dropdown Menü bereit, mit welchem es möglich ist verschiedene vorgefertigte Beispiele zu laden. Diese Beispiele werden automatisiert nach der Auswahl ans Backend geschickt und dort generiert, sodass nach einer kurzen Wartezeit ein Event Storming Board und falls vorhanden Mockups und Diagramme angezeigt werden können. Als Nächstes folgt ein Knopf zum Anstoßen einer Generierung, nachdem dieser Knopf gedrückt wurde und die Generierung angestoßen ist, erscheint ein Ladekreis in dem Knopf, um als Indikator dafür zu dienen, dass der Prozess noch nicht abgeschlossen ist. Die Web-Anwendung ermöglicht es zusätzlich die in der Oberfläche erstellten Dateien herun-

terladen zu können. Hierfür öffnet sich ein Pop-Up Bereich, nachdem der Download-Knopf betätigt wurde. Dieses Fenster ist in Abbildung 3.9 dargestellt.

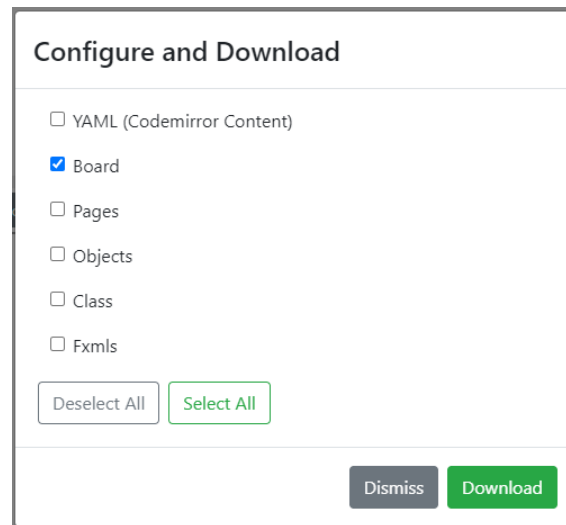


Abbildung 3.9: Download Fenster

Im sich öffnenden Fenster erhält der Benutzende die Möglichkeit auszuwählen, welche Dateien heruntergeladen werden sollen. Es können nur einzelne Dateien heruntergeladen werden, wie der Inhalt des Code-Editors, das generierte Event Storming Board oder aber alle Dateien die durch die aktuelle Workflow-Beschreibung im Code-Editor generiert wurden. Mit einem erneuten Klick auf den Download-Knopf, welcher sich neben dem Dismiss-Knopf befindet, wird eine Zip-Datei generiert und automatisch durch den jeweiligen Browser heruntergeladen. Für den Fall, dass der Nutzende noch keine Erfahrungen mit der Syntax von `fulibWorkflows` gemacht hat, kann die Dokumentation, welche auf Englisch verfasst ist, mit einem Klick auf den grauen Docs-Knopf geöffnet werden. Die Dokumentation stammt aus dem GitHub-Repository von `fulibWorkflows`. Im rechten Bereich der Navigationsleiste befinden sich zwei Buttons, welche das Theme des Code-Editors abändern, damit ist es möglich einen Light- oder Dark-Mode zum Schreiben des Codes zu verwenden. Daneben finden sich drei weitere Buttons, welche die Anzeige der generierten Dateien umschaltet. Hierbei kann zwischen dem Anzeigen der Mockups(Pages), der Objektdiagramme oder des Klassendiagramms umgeschaltet werden. Letztlich befindet sich ganz recht die aktuelle Versionsnummer des gesamten Web-Editors.

3.2.1 Code-Editor

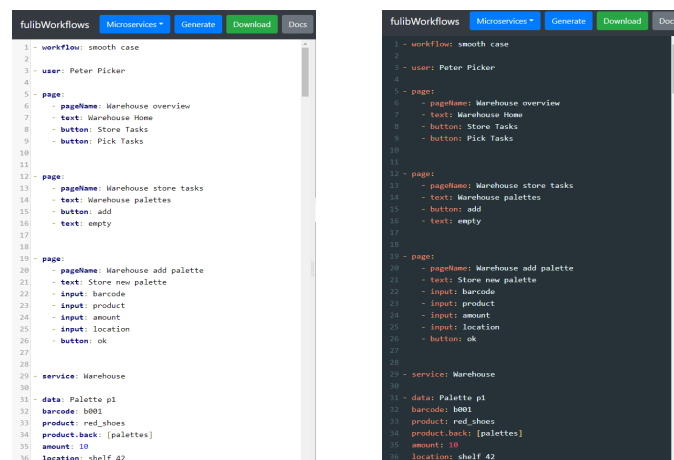
In dieser Sektion wird der Code-Editor, welcher das Herzstück der Anwendung ist erläutert. Dieser befindet sich auf der linken Seite der Oberfläche. Wie zuvor bereits beschrieben wurde hierbei ein Codemirror verwendet. Durch die Verwendung der `ngx-codemirror` Bibliothek vereinfacht sich das Einbinden eines Codemirrors in eine Angular Anwendung. Dadurch kann eine Konfiguration des Codemirrors über ein Options-Objekt übergeben werden. Die Konfiguration des Codemirrors ist in Listing 23 dargestellt.

```

49  this.codemirrorOptions = {
50    lineNumber: true,
51    theme: this.currentCodemirrorTheme,
52    mode: 'yaml',
53    extraKeys: {
54      'Ctrl-Space': 'autocomplete',
55      'Ctrl-S': generateHandler,
56    },
57    autofocus: true,
58    tabSize: 2,
59  };

```

Listing 23: Codemirror Konfiguration



(a) Light Theme

(b) Dark Theme

Abbildung 3.10: Themes des Codemirrors

In Zeile 50 werden die Zeilennummern am linken Rand des Codemirrors aktiviert. Das aktuelle Theme wird in der folgenden Zeile ebenfalls übergeben, da es zwei verschiedene Themes gibt, welche verwendet werden können, wird der Wert aus einer weiteren Variable übernommen. Codemirror stellt bereits zahlreiche verschiedene Themes bereit, für den Light-Mode wurde das *idea*-Theme verwendet, für den Dark-Mode das *material*-Theme. Initial wird der Codemirror mit dem Light-Theme geladen, das Umstellen des Themes erfolgt über die entsprechenden Knöpfe in der Navigationsleiste, wie bereits zuvor beschrieben. Abbildung 3.10 zeigt den Codemirror in beiden Modi, wobei sich sowohl Light- als auch Dark-Mode nah an den Standard-Modi von IntelliJ orientieren.

In Zeile 52 aus Listing 23 wird die Programmiersprache des Editors festgelegt. Da die Workflowbeschreibungen von fulibWorkflows in *.es.yaml*-Dateien angelegt/gespeichert werden, wird die Programmiersprache auf *yaml* festgelegt. Dieser Modus wird von Codemirror selbst bereitgestellt und Bedarf zur Verwendung lediglich eines Importes in der *main.ts* der Angular-Anwendung. Über die Option *extraKeys* können Tasten oder Tastenkombinationen an weitere Funktionen gekoppelt werden. In Zeile 54 wird das Öffnen einer Liste an

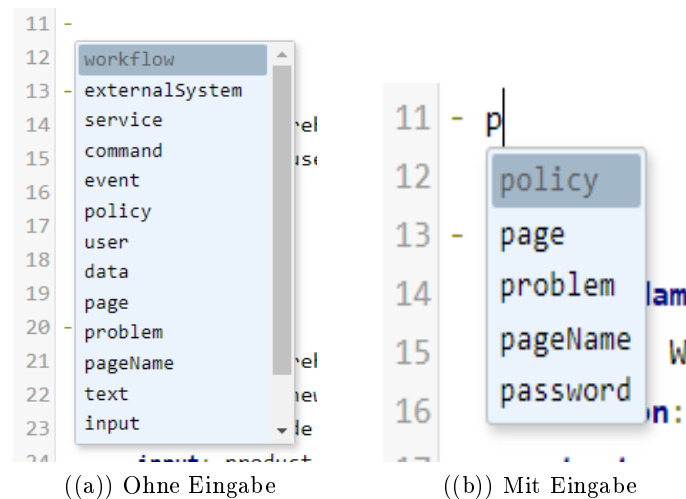


Abbildung 3.11: Autovervollständigung

vorgeschlagenen Wörtern geöffnet, nachdem der Nutzende ‘Strg+Leertaste’ gedrückt hat. Damit dies funktioniert benötigt es das Importieren des *show-hint*-AddOns von Codemirror in der *main.ts*-Datei.

Dieses Add-on erstellt die in Abbildung 3.11((a)) angezeigte Liste, der Inhalt der Liste wird über ein in dieser Arbeit erstelltes Codemirror Add-on gefüllt. Hierbei fällt auf, dass auch Schlüsselwörter angezeigt werden, welche nur im Kontext eines Page-Notes Sinn ergibt. Das eigens geschriebene Add-on ist nicht kontextsensitiv und besitzt somit nicht die gleichen Funktionen wie die Autovervollständigung, welche eine IDE über das JSON-Schema erstellt. Der Code des geschriebenen Add-ons befindet sich im Anhang dieser Arbeit, hierbei wird über die aktuelle Position des Cursors, das aktuelle Wort extrahiert. Damit ist es möglich über die Liste der Schlüsselwörter zu iterieren und zu prüfen, welche Vorschläge sinnvoll sind, sollte ein Wort bereits begonnen sein. Dies ist anhand von Abbildung 3.11((b)) genauer zu erkennen. Hierbei wurde bereits der Buchstabe *p* eingetippt und das Add-on bietet zur Vervollständigung lediglich Wörter an, welche mit *p* beginnen.

Des Weiteren wird durch das Betätigen der Tastenkombination ‘Strg+S’ die Generierung angestoßen. Bevor die Daten aus dem Codemirror zur Generierung an das Backend gesendet werden, werden diese auf Richtigkeit überprüft. Da bereits ein JSON-Schema für *fulibWorkflows* existiert, wurde eine Bibliothek ausgewählt, welche einen Text über ein JSON-Schema validieren kann. *Ajv* ist ein solcher Validierungsmechanismus, allerdings kann mit *Ajv* lediglich ein JSON-Objekt mittels JSON-Schema validiert werden.[Pob21] Somit wurde zum Umwandeln des Textes aus dem Codemirror zu einem JSON-Objekt die Bibliothek *js-yaml* verwendet.[Zap21]

Sobald während der Validierung ein Problem mit der Eingabe erkannt wird, gibt diese einen Fehler zurück. Dieser Fehler wird anschließend als Toast in der Oberfläche angezeigt um den Nutzenden darauf hinzuweisen, an welcher Stelle die Eingabe im Codemirror nicht dem JSON-Schema von *fulibWorkflows* entspricht. Eine solche Fehlermeldung ist in Ab-

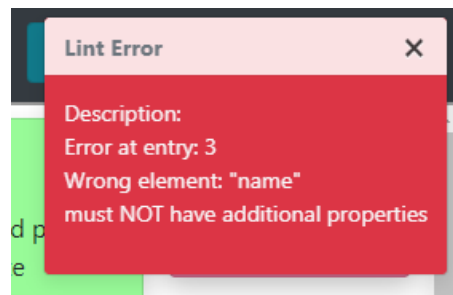


Abbildung 3.12: Validierungerror als Toast

Abbildung 3.12 dargestellt. Hierbei wurde einem Note ein Attribut *name* zugeordnet. Da der besagte Note allerdings ein *user* ist und im JSON-Schema festgelegt ist, dass ein user Note keine zusätzlichen Attribute besitzen darf, entsteht der Fehler aus der Abbildung. Diese Fehlermeldung wird nach 20 Sekunden automatisch geschlossen. Für die Darstellung von Toasts wurde die gleichnamige Komponente von *ng-bootstrap* verwendet. Sollte die Eingabe valide sein, so wird diese über einen Service an das Backend geschickt.

3.2.2 Darstellung generierter Dateien

Die anderen beiden Bereiche der Oberfläche, welche bisher nicht erläutert wurden, stellen beide jeweils einen bestimmten Teil der generierten Dateien dar. Beide Bereiche bestehen aus einem IFrame, wobei der obere IFrame lediglich das generierte Event Storming Board anzeigt und der untere IFrame die Anzeige von HTML-Mockups und Klassen-/Objektdiagrammen übernimmt.

```

1  export interface GenerateResult {
2    board: string,
3    pages: Map<number, string>,
4    numberOfPages: number,
5    diagrams: Map<number, string>,
6    numberOfDiagrams: number,
7    classDiagram: string,
8  }

```

Listing 24: Modell der vom Backend empfangenen Daten

Bevor auf die Darstellung und Funktionen der Iframes eingegangen wird, wird zuerst betrachtet in welcher Form die generierten Dateien vom Backend im Frontend verarbeitet werden. Die Form ist in Listing 24 dargestellt, in einem *GenerateResult*-Objekt werden alle generierten Dateien und Zusatzinformationen abgespeichert. Bei den Zusatzinformationen handelt es sich um die Anzahl der generierten Diagramme und HTML-Mockups. Die generierten Dateien werden lediglich als reiner String behandelt. Um den Zugriff auf einzelne Objektdiagramme oder Mockups zu vereinfachen, sind diese jeweils in einer Map abgespeichert. Bei den Maps ist einer Nummer ein Diagramm/Mockup zugeordnet.

Der obere IFrame erhält als Eingabe das Event Storming Board und stellt dieses dar. Dies ist möglich, da das Event Storming Board eine valide HTML-Datei ist, welche durch einen IFrame dargestellt werden kann. Hierbei war es nötig eine Pipe zu erstellen, welche den *DomSanitizer* von Angular auf der Eingabe umgeht.[**safe-pipe**] Der DomSanitizer ist ein von Angular bereitgestellter Service, welcher Elemente aus der Eingabe entfernt die potenziell für Angreifer genutzt werden könnten, um Skripte auf der Anwendung auszuführen. Dies ist ein Sicherheitsmechanismus um das sogenannte *Cross-site scripting* auszuhebeln. Beim *Cross-site scripting* können Angreifer JavaScript Code in den Browsern anderer Nutzer ausführen und somit personenbezogene Daten erhalten.[**xss**] Durch die Pipe wird dieser Sicherheitsmechanismus umgangen und die Eingabe wird unverändert im IFrame geladen.

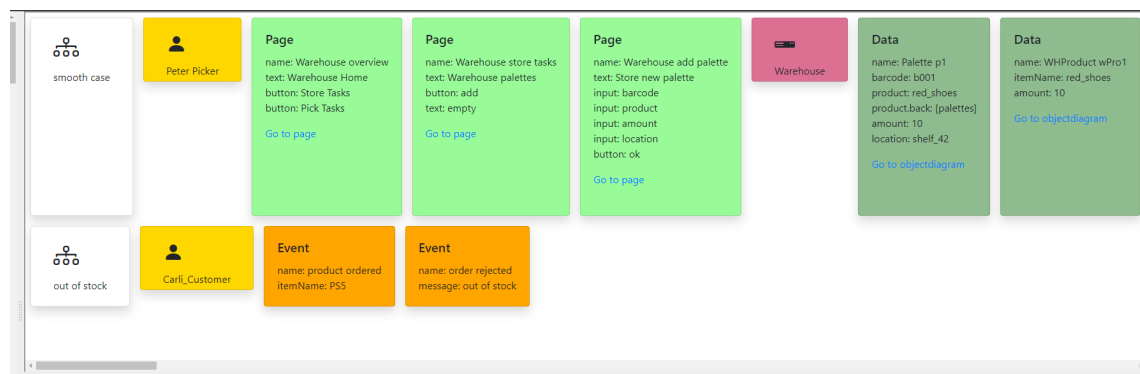


Abbildung 3.13: Event Storming Board in einem IFrame

In Abbildung 3.13 ist ein Event Storming Board für das im Web-Editor vorhandene Microservices Beispiel dargestellt. Hierbei ist zu erwähnen, dass die Bereiche: Code-Editor, Event Storming Board IFame und der IFrame zur Anzeige der Mockups und Diagramme beliebig vergrößert oder verkleinert werden kann. Um dies zu ermöglichen wurde die Bibliothek *angular-split* verwendet. Diese ermöglicht es Bereiche zu definieren und darin Inhalt zu platzieren, sowie die Veränderung der Größen der Bereiche bereitzustellen.[**angular-split**] Ohne diese Funktionalität können auch größere Boards in der Oberfläche angezeigt werden.

Für den unteren IFrame gelten die gleichen gegebenheiten wie für den oberen IFrame. Da es mehrere Mockups oder Diagramme geben kann, existieren vier Knöpfe, mit welchen es möglich ist zwischen den Mockups/Diagrammen zu wechseln. Doch der Wechsel zwischen diesen Elementen ist nicht nur über die vier Knöpfe möglich, sondern ebenfalls über die entsprechenden Notes aus dem Board IFrame. Für jeden Page oder Data Note existiert in der Anzeige ein Link, welcher als Knopf fungiert, mit welchem zu dem Mockup oder Diagramm des Notes gewechselt werden kann.

```
1  (<any>window).setIndexFromIframe = this.setIndexFromIframe.bind(this);
```

Listing 25: Bereitstellung einer Methode

Hierfür ist eine Methode erstellt worden, welche für die gesamte Oberfläche sichtbar ist. In der Generierung mittels `fulibWorkflows` wird dieser Link erstellt, darin wird die bereitgestellte Methode mittels `window.parent.setIndexFromIframe(0);` aufgerufen. Hierbei wird über das Fenster auf die oberste Komponente der Anwendung zugegriffen, dort ist durch den Code aus Listing 25 die Methode `setIndexFromIframe` bereitgestellt.

3.3 fulibWorkflows Web-Editor Backend

Das Backend des Web-Editors basiert auf einem mit Spring Initializr generiertem Java Projekt. Zusätzlich wurden bei der Konfiguration die Dependencies für eine Spring Web Anwendung hinzugefügt. Neben den eben genannten Dependencies wurde lediglich `fulibWorkflows` als weitere Bibliothek zum Backend hinzugefügt.

Die verfügbaren Endpunkte des Backends werden in einem Controller bereitgestellt. In diesem Fall ist dies der `FulibWorkflowsController`, welcher mit zwei Annotations versehen ist. Die erste Annotation ist `@Controller`, welche der Anwendung mitteilt, dass diese Klasse als Controller agiert. Bei der zweiten Annotation handelt es sich um `@CrossOrigin()` mit welcher es ermöglicht wird problemlos mit dem Frontend zu interagieren. Im `FulibWorkflowsController` sind Endpunkte für die Generierung und den Download definiert. Diese Definitionen werden in Listing 26 zur Veranschaulichung dargestellt.

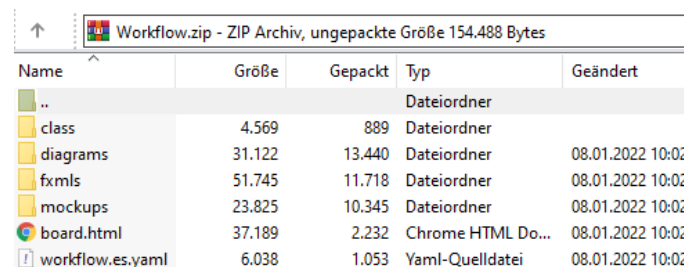
```
15  @PostMapping(path = "/generate", consumes = MediaType.ALL_VALUE)
16  @ResponseBody
17  public String generate(@RequestBody String yamlData) {
18      return fulibWorkflowsService.generate(yamlData);
19  }
20
21  @PostMapping(path = "/download",
22              consumes = MediaType.ALL_VALUE,
23              produces = "application/zip")
24  @ResponseBody
25  public byte[] download(@RequestBody String yamlData,
26                       @RequestParam Map<String, String> queryParams) {
27      return fulibWorkflowsService.createZip(yamlData, queryParams);
28  }
```

Listing 26: Definition der Endpunkte

Beide Endpunkte können mit einem POST-Request angesprochen werden, da sowohl beim Download als auch der Generierung der Inhalt der yaml Datei vom Frontend mitgeschickt wird. Dies sorgt dafür, dass bei beiden Endpunkten die Generierung durchgeführt wird, um kein Abspeichern von Dateien und Vergabe von IDs für eine Beschreibung angelegt und verwaltet werden muss. Damit die Endpunkte auf diesen Inhalt zugreifen können, ist der erste Methodenparameter mit der `@RequestBody` Annotation versehen. Hierbei wird

auf den Inhalt zugegriffen, welcher vom Frontend in der Post-Anfrage mitgesendet wurde. Der Download-Endpunkt erhält zusätzlich zu der yaml Beschreibung ebenfalls Query-Parameter. Diese enthalten die aus dem Download-PopUp des Frontends angegebenen Dateien, welche heruntergeladen werden sollen. Mittels der `@ResponseBody` Annotation, können der Antwort des Backends weitere Daten hinzugefügt werden, in welcher Form diese sind, resultiert aus dem Rückgabetyt der jeweiligen Methode unterhalb der Annotation. Bei der Generate-Methode wird ein JSON-Objekt als String zurückgegeben, wobei bei der Download-Methode das generierte Zip-Archiv als ByteArray verwendet wird. Beide Methoden des Controllers reichen die erhaltenen Daten an den `FulibWorkflowsService` weiter, welcher sich um die Generierung über `fulibWorkflows` und der Erstellung des Zip-Archives kümmert.

Im `FulibWorkflowsService`, wird sowohl in der `generate`- als auch der `createZip`-Methode, welche vom Controller aufgerufen werden, der yaml-String über `fulibWorkflows` generiert. Dabei nutzt das Backend die `generateAndReturnHTMLsFromString`-Methode des Board-Generators von `fulibWorkflows`. Im Anschluss wird aus der Map von generierten Dateien ein `GenerateResult` erstellt, welches von der `generate`-Methode zu einem JSON-String weiterverarbeitet und zurück an den Controller gibt. Nachdem das `GenerateResult`-Objekt in der `createZip`-Methode erstellt wurde, beginnt das Erstellen des Zip-Archivs. Auf Grundlage der Query-Parameter werden lediglich die Dateien zum Zip-Archiv hinzugefügt, welche im Frontend ausgewählt werden. In Abbildung 3.14 sollen alle Dateien heruntergeladen werden, welche von `fulibWorkflows` generiert wurden.



Name	Größe	Gepackt	Typ	Geändert
..			Dateiordner	
class	4.569	889	Dateiordner	
diagrams	31.122	13.440	Dateiordner	08.01.2022 10:02
fxmls	51.745	11.718	Dateiordner	08.01.2022 10:02
mockups	23.825	10.345	Dateiordner	08.01.2022 10:02
board.html	37.189	2.232	Chrome HTML Do...	08.01.2022 10:02
workflow.es.yaml	6.038	1.053	Yaml-Quelldatei	08.01.2022 10:02

Abbildung 3.14: Inhalt eines heruntergeladenen Zip-Archivs

Um die Dateien strukturiert zu halten, werden Diagramme und Mockups in eigenen Unterordner abgelegt. Hierbei wird zwischen Objektdiagrammen und dem Klassendiagramm, als auch zwischen HTML- und FXML-Mockups unterschieden. Auf oberster Ebene werden sowohl die Workflowbeschreibung als auch das generierte Event Storming Board abgelegt, da diese die Grundlage für alle weiteren Dateien legen. Das Klassendiagramm ist im Ordner *class* abgelegt, die Objektdiagramme im *diagrams*-Ordner. Im Gegensatz hierzu, werden die HTML-Mockups im *mockups*-Ordner hinterlegt, die gleichen Mockups im FXML-Format werden im *fxmls*-Ordner platziert.

4 Evaluation

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

4.2 Auswertung

TODO: this

Was ist die Meinung des Experten zu der Anwendung?

Kann ihm das helfen? (Begründung durch seine Vorerfahrungen)

5 Fazit

TODO: this

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

TODO: this

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.
- [Coo21] Scott Cooper. *ngx-codemirror*. 2021. URL: <https://github.com/scttcper/ngx-codemirror>.
- [Goo21] Google. *Introduction to the Angular Docs*. 2021. URL: <https://angular.io/docs>.
- [Hav21] Marijn Haverbeke. *CodeMirror*. 2021. URL: <https://codemirror.net/>.
- [Kas21a] Fujaba Team Kassel. *fulib - Fujaba library*. 2021. URL: <https://github.com/fujaba/fulib#fulib---fujaba-library>.
- [Kas21b] Fujaba Team Kassel. *fulibTools - Additional features for fulib*. 2021. URL: <https://github.com/fujaba/fulibTools#fulibtools---additional-features-for-fulib>.
- [Kri21] Mads Kristensen. *JSON Schema Store*. 2021. URL: <https://www.schemastore.org/json/>.
- [Kün21] Maximilian Freiherr von Künßberg. *pm.es.yaml*. 2021. URL: <https://raw.githubusercontent.com/fujaba/fulibWorkflows/main/src/gen/resources/pm.es.yaml>.
- [Pob21] Evgeny Poberezkin. *Ajv JSON schema validator*. 2021. URL: <https://ajv.js.org/>.
- [Tea21a] Bootstrap Team. *Bootstrap Icons*. 2021. URL: <https://icons.getbootstrap.com/>.
- [Tea21b] Bootstrap Team. *Build fast, responsive sites with Bootstrap*. 2021. URL: <https://getbootstrap.com/>.

- [Tea21c] JSON Schema Team. *Getting Started Step-By-Step*. 2021. URL: <https://json-schema.org/learn/getting-started-step-by-step#defining-the-properties>.
- [VMw21] Inc. VMware. *spring initializr*. 2021. URL: <https://start.spring.io/>.
- [Zap21] Alexey Zapparov. *js-yaml*. 2021. URL: <https://github.com/nodeca/js-yaml>.
- [Cla22] ClaudRepo. *Public Maven Repositories: Maven Central and More*. 2022. URL: <https://www.cloudrepo.io/articles/public-maven-repositories-maven-central-and-more.html>.
- [Sal22a] Salesforce.com. *Language Support*. 2022. URL: <https://devcenter.heroku.com/categories/language-support>.
- [Sal22b] Salesforce.com. *What is Heroku?* 2022. URL: <https://www.heroku.com/what>.
- [VMw22a] Inc. VMware. *Spring Boot*. 2022. URL: <https://spring.io/projects/spring-boot>.
- [VMw22b] Inc. VMware. *Why Spring?* 2022. URL: <https://spring.io/guides/gs/rest-service/>.

8 Anhang

8.1 fulibWorkflows JSON-Schema

TODO: This

```
{
  "$schema": "https://json-schema.org/draft-07/schema#",
  "title": "JSON schema for fulibWorkflows ",
  "description": "Schema for the generation file of fulibWorkflows",
  "type": "array",
  "additionalItems": false,
  "$defs": {
    "workflowItem": {
      "description": "The title of your current workflow",
      "type": "object",
      "properties": {
        "workflow": {
          "type": "string"
        }
      }
    },
    "required": [
      "workflow"
    ],
    "additionalProperties": false
  },
  "externalSystemItem": {
    "description": "Can be used to address data coming from another source",
    "type": "object",
    "properties": {
      "externalSystem": {
        "type": "string"
      }
    }
  },
  "required": [
    "externalSystem"
  ],
  "additionalProperties": false
},
  "serviceItem": {
    "description": "The service on which the following events are executed",
    "type": "object",
    "properties": {
      "service": {
        "type": "string"
      }
    }
  },
  "required": [
    "service"
  ],
  "additionalProperties": false
},
  "commandItem": {
    "description": "A command send by a user",
    "type": "object",
    "properties": {
      "command": {
        "type": "string"
      }
    }
  },
  "required": [
    "command"
  ],
  ],
}
```

```
{
  "$schema": "https://json-schema.org/draft-07/schema#",
  "title": "JSON schema for pages used in fulibWorkflows ",
  "description": "Page Schema for the generation file of fulibWorkflows",
  "type": "array",
  "additionalItems": false,
  "$defs": {
    "nameItem": {
      "description": "Defines the name of the page",
      "type": "object",
      "properties": {
        "pageName": {
          "type": "string"
        }
      }
    },
    "required": [
      "pageName"
    ],
    "additionalProperties": false
  },
  "textItem": {
    "description": "Defines a text",
    "type": "object",
    "properties": {
      "text": {
        "type": "string"
      }
    }
  },
  "required": [
    "text"
  ],
  "additionalProperties": false
},
  "buttonItem": {
    "description": "Defines a button with optional command",
    "type": "object",
    "properties": {
      "button": {
        "type": "string"
      }
    }
  },
  "required": [
    "button"
  ],
  "additionalProperties": false
},
  "inputItem": {
    "description": "Defines an input with optional fill",
    "type": "object",
    "properties": {
      "input": {
        "type": "string"
      }
    }
  },
  "required": [
    "input"
  ],
}
```

```

grammar FulibWorkflows;

// Non-Terminals

file: workflows+ ;

workflows: workflow NEWLINE eventNote* ;

eventNote: ( normalNote | extendedNote | page) NEWLINE? ;

workflow: MINUS 'workflow' COLON NAME ;

normalNote: MINUS NORMALNOTEKEY COLON NAME ;

extendedNote: MINUS EXTENDEDNOTEKEY COLON NAME NEWLINE attribute* ;

page: MINUS 'page' LISTCOLON NEWLINE pageList ;

attribute: INDENTATION NAME COLON value NEWLINE? ;

value: NAME | NUMBER | LIST;

pageList: pageName NEWLINE element* ;

pageName: INDENTATION MINUS 'pageName' COLON NAME ;

element: INDENTATION MINUS ELEMENTKEY COLON NAME NEWLINE ;

// Terminals
NORMALNOTEKEY: 'externalSystem' | 'service' | 'command' | 'policy' | 'user' | 'problem'

EXTENDEDNOTEKEY: 'event' | 'data' ;

ELEMENTKEY: 'text' | 'input' | 'password' | 'button' ;

NAME: ([A-Za-zÄÖÜß] [0-9]* [-/_,. '@!']* [ ]* [0-9]* [-/_,. '@!']* [ ]*)+ ;

MINUS: '-' ;

COLON: ':' ;

LISTCOLON: ':' ;

KEY: [A-Za-z.]+ ;

NEWLINE: [\r\n]+ | [\n]+ ;

LIST: '[' (.) *? ']' ;

NUMBER: [0-9]+ ;

INDENTATION: [\t]+ | [ ]+ ;

```

Listing 29: fulibWorkflows Antlr Grammatik


```

import * as CodeMirror from 'codemirror';

const noteKeywords = ['workflow', 'externalSystem', 'service', 'command', 'event', 'poli'];
const pageKeywords = ['pageName', 'text', 'input', 'password', 'button'];

// @ts-ignore
CodeMirror.registerHelper('hint', 'fulibWorkflows', (cm) => {
  const cur = cm.getCursor();
  const range = cm.findWordAt(cur);
  let start = range.anchor.ch;
  let end = range.head.ch;

  // Get Current Word
  const word = cm.getRange(range.anchor, range.head);

  // Filter Keywords for possible completions for the current word
  const result: string[] = [];
  for (const noteKeyword of noteKeywords) {
    if (!word || word === ' ' || noteKeyword.indexOf(word) === 0) {
      result.push(noteKeyword);
    }
  }

  for (const pageKeyword of pageKeywords) {
    if (!word || word === ' ' || pageKeyword.indexOf(word) === 0) {
      result.push(pageKeyword);
    }
  }

  return {
    list: result,
    from: CodeMirror.Pos(cur.line, start),
    to: CodeMirror.Pos(cur.line, end),
  };
});

CodeMirror.commands.autocomplete = (cm) => {
  // @ts-ignore
  CodeMirror.showHint(cm, CodeMirror.hint.fulibWorkflows);
}

```

Listing 30: Codemirror Add-on für Vervollständigung