Modularisierung und Zentralisierung von Angular-Komponenten auf Basis des Atomic Design

**T1000**

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Jonas Scherer

07.09.2021

Bearbeitungszeitraum 12 Wochen

Matrikelnummer, Kurs 6499975, TINF20B

Dualer Partner DaimlerTSS, Stuttgart

Betreuer\*in des Dualen Partners Herr Johannes Frey

# Inhaltverzeichnis

[Inhaltverzeichnis 2](#_Toc70588232)

[Abbildungsverzeichnis 5](#_Toc70588233)

[Tabellenverzeichnis 6](#_Toc70588234)

[Listings 7](#_Toc70588235)

[1 Problemstellung, Ist-Zustand Soll-Zustand 8](#_Toc70588236)

[2. Vor- und Nachteile einer Zentralisierung 9](#_Toc70588237)

[3 Grundlagen, Stand der Technik 10](#_Toc70588238)

[3.1 Atomic Design Pattern 10](#_Toc70588239)

[3.1.1 Atome 10](#_Toc70588240)

[3.1.2 Moleküle 11](#_Toc70588241)

[3.1.3 Organismen 11](#_Toc70588242)

[3.1.4 Templates 12](#_Toc70588243)

[3.1.5 Pages 12](#_Toc70588244)

[3.2 GitHub 13](#_Toc70588245)

[3.3 Angular 14](#_Toc70588246)

[3.4 Typescript 16](#_Toc70588247)

[4. Bereitstellen eines zentralen Repositorys 17](#_Toc70588248)

[5. Anpassung des Build-Workflows 18](#_Toc70588249)

[6. Kategorisieren einzelner Komponenten 19](#_Toc70588250)

[6.1 Atome 19](#_Toc70588251)

[6.2 Moleküle 22](#_Toc70588252)

[6.2.1 Info-Dialog Komponente 22](#_Toc70588253)

[6.2.2 Top-Bar Komponente 23](#_Toc70588254)

[6.2.3 License-Dialog Komponente 24](#_Toc70588255)

[6.3 Organismen 25](#_Toc70588256)

[6.3.1 Copyright-Footer Komponente 25](#_Toc70588257)

[6.4 Pages 26](#_Toc70588258)

[6.4.1 Page-Not-Found Komponente 26](#_Toc70588259)

[7. Zentralisierung von Angular-Material Themes 27](#_Toc70588260)

[8. Zentralisierung von CSS-Styles 28](#_Toc70588261)

[Literaturverzeichnis 28](#_Toc70588262)

Abkürzungsverzeichnis

Git

Repo

SSH

# Abbildungsverzeichnis

Abbildung 8: https://github.com/

# Tabellenverzeichnis

# 

# Listings

# 1 Problemstellung, Ist-Zustand Soll-Zustand

Betroffen dieser Zentralisierung und Modularisierung sind drei bereits existierende, voneinander autark agierende Userinterfaces, welche in einem gemeinsamen GitHub Repository liegen. Aufgrund des Sourcecode Umfangs dieser drei Userinterfaces, ihrer Kopplung und ihrer monolithischen Softwarestruktur, sollen sich stark ähnelnde Komponenten dieser zentralisiert werden, sodass eine Reduktion von Sourcecode sowie eine einheitliche User Experience erzielt werden kann. Zur Kategorisierung der einzelnen Komponenten soll das Atomic Design Pattern angewendet werden, um eine klar definierte Reihenfolge der zu Zentralisierten Komponenten aufstellen zu können. Ohne diese Reihenfolge kann dieses Vorhaben zu mehr Schaden als Nutzen führen.

Da die Userinterfaces auf dem Framework Angular und der Programmiersprache Typescript basieren, soll der technologische Fokus auf diese gelegt werden. Die Module erfordern nicht nur eine Separierung im Code, sondern auch auf der Ebene der Source Code Verwaltung. Hier gilt es ein zentrales Repository bereitzustellen, aus dem die Komponenten konsumiert werden. Mit dieser Erweiterung muss außerdem eine Integration der Separierung in den Build-Workflow des Endsystems durchgeführt werden.

Die relevanten Komponenten werden konkret angepasst, modularisiert und zentralisiert.

# 2. Vor- und Nachteile einer Zentralisierung

Die Zentralisierung von Softwarecode bringt eine Menge Vorteile mit sich. Nicht nur die Möglichkeit, gleiche Funktionalität bei weniger Sourcecode beizubehalten, sondern kann durch diese auch über mehrere Userinterfaces hinweg ein einheitlicher Style eingehalten und so eine einheitliche User Experience gewahrt werden. Außerdem ist man in der Lage Änderungen in einer einzelnen Komponente vorzunehmen, welche sich direkt auf alle Userinterfaces auswirken, welche diese Komponente eingebunden haben. Mit dieser Vorgehensweise, kann man sich von der Monolithischen Struktur eines Softwarecodes hervorragend distanzieren.

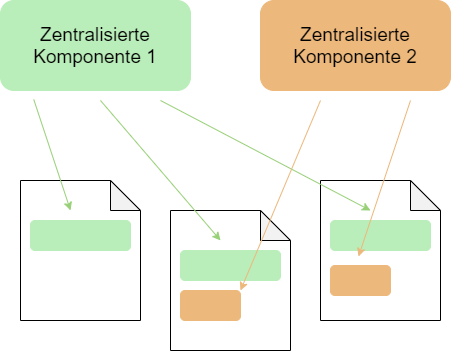


Abbildung 1: Visualisierung der Zentralisierung

Das Auslagern von Softwarecode kann aber auch ein Nachteil sein. So schön es klingt, dass nur eine Änderung in einer zentralisierten Komponente vorgenommen werden muss und dies sich auf alle Userinterfaces auswirkt, genauso gefährlich ist dies auch. Durch eine kleine Änderung in einer Datei können unzählige Fehler auftreten.  
Weitere stark überwiegende Nachteile birgt die Zentralisierung nicht. Hat man einmal die richtigen Tools, wie hier GitHub, Angular, Typescript, bereits existierende Userinterfaces und ein Buildsystem, so steht der Zentralisierung nichts mehr im Wege.

# 3 Grundlagen, Stand der Technik

## 3.1 Atomic Design Pattern

Das Atomic Design Pattern nach Brad Frost, beschreibt die Zerlegung von komplexem Softwarecode in kleine, für jeden Softwareengineer nachvollziehbare Elemente. So wird aus einem enormen Code für ein Userinterface ein deutlich schlankerer, welcher einzelne Komponenten aufruft. Diese wiederrum können auch wieder zerlegt werden.

Das von Brad Frost entwickelte Pattern findet seinen Ursprung in der Chemie. Er beschreibt die Hinführung und den Gedankengang, welcher zur Entwicklung dieses Design Patterns beitrugen, in seinem Buch „Atomic Design“, welches 2016 erschienen ist, genauer. Faszinierend ist aber nicht die Geschichte, sondern die Theorie sowie die Anwendung des Atomic Designs.

Dieses Design unterteilt Softwarecode in fünf Kategorien, nämlich in:

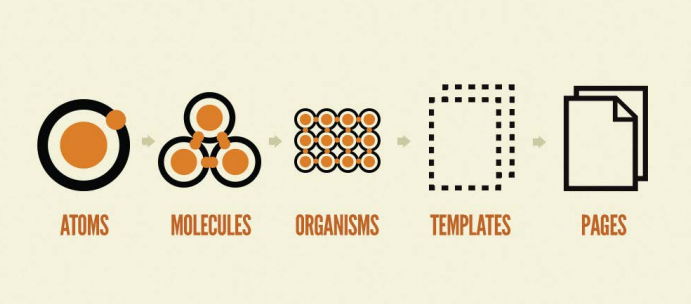


Abbildung 2: Atomic Design Pattern

### 3.1.1 Atome

Selbstverständlich gibt es in der Chemie kleinere Einheiten als ein Atom, beispielsweise Neutronen oder Protonen. Das Atom aber, ist die kleinste funktionelle Einheit, und somit stellt es auch im Atomic Design, die kleinstmögliche Einheit da.

Ein Atom, nach Brad Frost, enthält grundlegende HTML Elemente wie beispielweise einen Knopf zum Schließen eines Fensters oder ein Label in welches etwas hineingeschrieben wird.[[1]](#footnote-1)1) Zu Atomen zählen aber auch die verschiedenen Styles eines jeweiligen Elements, zum Beispiel die Farbe und Größe eines Knopfes.

### 3.1.2 Moleküle

In der Chemie ist ein Molekül ein Zusammenschluss aus mindestens zwei Atomen, was auch für das Atomic Design Pattern gilt. Das Beispiel Molekül in Abbildung 2 enthält einen Knopf zum Schließen des Fensters und ein Label, in welchem der Fülltext steht. Somit besteht dieses Molekül aus zwei Atomen.

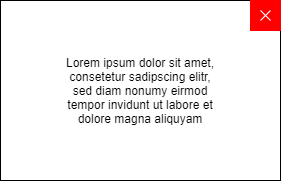


Abbildung 3: Beispiel Molekül

Werden zwei verschiedene Atome kombiniert, so erlangen diese einen Sinn.[[2]](#footnote-2)2) Schon jetzt zeigt sich deutlich, inwiefern dieses Prinzip im späteren Verlauf von Nutzen sein kann

### 3.1.3 Organismen

Ein Organismus ist ein schon deutlich komplexeres Softwareelement, denn sie können aus verschiedensten Zusammensetzungen von Atomen und Molekülen bestehen. Auch können sie das gleiche Atom oder Molekül beliebig oft beinhalten. Ein perfektes Beispiel, dessen sich auch Brad Frost in seinem Buch „Atomic Design“ bedient, ist die Kopfzeile, bzw. der sogenannte Header einer Website.



Abbildung 4: Beispiel Header

Der Beispiel Header in Abbildung 3 ist ein Beispiel für eine sehr einfache Kopfzeile einer Website. Dort befinden sich, von links betrachtet, drei Knöpfe und rechts davon ein Molekül. Dieses Molekül besteht aus einem Eingabefeld und einem Knopf. Somit ist hier eine steigende Komplexität festzustellen

### 3.1.4 Templates

Templates beziehungsweise Vorlagen kommen nicht mehr in der Chemie vor, sie wurden von Brad Frost in diese Reihe gegliedert. Sie beschreiben den grundlegenden Aufbau einer Webseite zum Beispiel welche Komponenten an welche Stelle eingefügt werden. Sie beachten aber nicht, welchen Inhalt eine Komponente besitzt.

„You can create good experiences without knowing the content. What you can’t do is create good experiences without knowing your content structure”[[3]](#footnote-3)3), ein Zitat von Mark Boulton, welches die Wichtigkeit eins guten Template ausdrückt.

### 3.1.5 Pages

Pages beziehungsweise Seiten wurden ebenfalls von Brad Frost selbst in die Reihe des Atomic Design Patterns gegliedert. Sie werden deklariert als spezifische Instanz einer Vorlage, welche mit Inhalt gefüllt wird. Sie zeigt dann dem Entwickler, wie die Webseite final aussehen wird. Mit genau dieser Webseite interagieren dann die Endnutzer.[[4]](#footnote-4)4)

## 3.2 GitHub

„GitHub ist ein webbasierter kollaborativer Hosting-Dienst für […]“[[5]](#footnote-5)5) Softwareprojekte. Durch diesen Dienst wird mehr als 56 Millionen Entwicklern und mehr als drei Millionen Organisationen die Möglichkeit gegeben, gemeinsam und ortsunabhängig an einem oder mehreren Projekten zu arbeiten.[[6]](#footnote-6)6) Außerdem wird man als Entwickler nicht von der Arbeit anderer eingeschränkt, da jeder vorerst lokal auf seinem Endgerät arbeitet.

Die Kombination aus Repositorys und der Verwaltung von Version verschiedenster Dateien macht GitHub erst so Interessant für Unternehmen, welche mehrere Mitarbeiten an einem Projekt beschäftigen. Ein Repository ist sozusagen ein Ordner, auf welchen ein Mitarbeiter eines Projekts Zugriff hat. Dieses kann er sich klonen und hat den neusten Stand dieses Repos dann lokal auf seinem Endgerät und kann dort arbeiten ohne andere Entwickler dieses Projekts zu beeinflussen. Ist er fertig mit seiner Arbeit, so kann er diese in das Repo in der Cloud wieder ‚pushen‘, womit jeder Entwickler dann Zugriff auf die Änderungen hat. Jedes Repository hat einen sogenannten Masterbranch, in welchem das Hauptprojekt verläuft. Jeder Entwickler kann von diesem Branch, einen eigenen Brach abzweigen und auf diesem beispielsweise ein neues Feature entwickeln. Ist dieses Feature fertig, so kann er seinen Featurebranch pushen. Möchten seine Mitarbeiter dieses Feature im Projekt haben, so können sie diesen Featurebranch wieder zurück in das Projekt führen und alle Änderungen werden auf den Masterbranch übernommen.

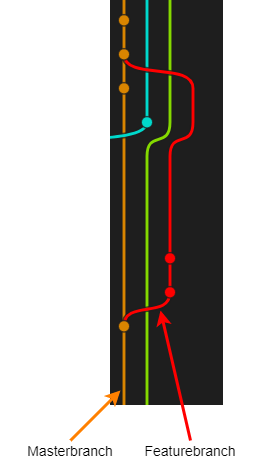


Abbildung 5: Gitbranching

In diesem Projekt führt an einer Versionsverwaltung kein Weg vorbei da die Zentralisierung von Komponenten an schon bestehenden oder noch mitten in der Entwicklung steckenden Userinterfaces durchgeführt wird und somit mehrere Entwickler zeitgleich an diesen arbeiten.

## 3.3 Angular

Der Nachfolger Angular von AngularJS ist ein Open Source Front-End-Webapplikationsframework, welches von Google LLC 2016 entwickelt wurde und auf Typescript basiert. Die Besonderheit von Angular ist der Aufbau einer Webapplikation in einzelnen Komponenten welche eine festgelegte Hierarchie befolgen.

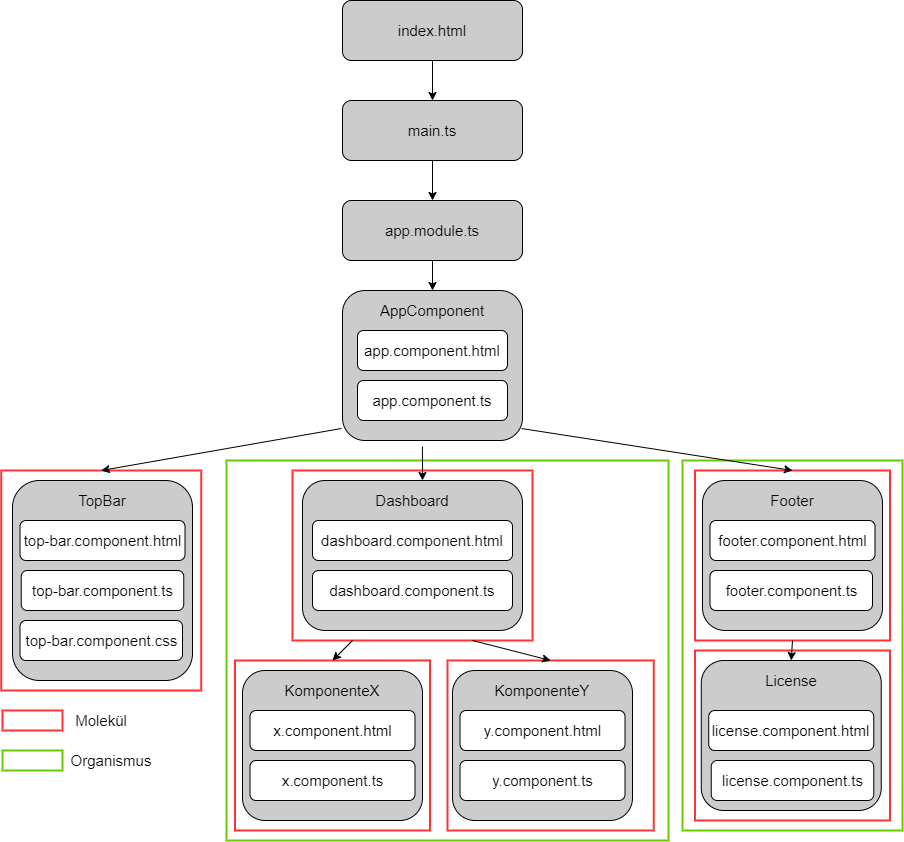


Abbildung 6: Beispielhafte Angular Struktur

Die index.html stellt das HTML-Grundgerüst dar, auf welchem die main.ts aufsetzt, die Startdatei für das Bootstrapping. In der darauffolgenden Typescript Datei app.module.ts, dem zentralen Angular-Modul, müssen alle Komponenten welche das Projekt enthält und benutzt importiert und deklariert werden, um diese nutzen zu können. Ab diesem Punkt befindet man sich schon mitten in der Webapplikation, denn in der AppComponent befindet sich das Template (app.component.html) sowie die Root-Komponente (app.component.ts) der Webapplikation. In Abbildung 6 ist anschaulich, dass aus diesen Dateien im Anschluss alle zu verwendenden Angular Komponenten aufgerufen werden, ob direkt, oder indirekt. Hier kann es sich bei einzelnen Komponenten um Moleküle oder gar Organismen handeln.[[7]](#footnote-7)7)

Atome befinden sich normalerweise innerhalb der einzelnen Komponenten beziehungsweise Molekülen. Enthält die TopBar beispielsweise nur einen Knopf und einen Text, so ist sie ein Molekül. Nutzt eine Komponente aber eine andere, so ist diese Einheit ein Organismus, sie ist deutlich komplexer als ein einfaches Molekül. Stylings der einzelnen Atome innerhalb der Moleküle können sich entweder in der jeweiligen HTLM-Datei oder in einer weiteren CSS-Datei befinden.

Um eine Webapplikation lokal auf einer Adresse, beispielweise localhost:4200/xy zu testen, bietet Angular-CLI den Befehl ng serve, welcher im jeweiligen Projektordner ausgeführt werden muss. Dies bietet die Möglichkeit in Echtzeit seinen Code zu kompilieren und zu testen, was im Bereich der Front-End-Entwicklung zu einem angenehmeren Workflow beiträgt.

## 3.4 Typescript

Die noch junge Programmiersprache Typescript wurde 2012 von Microsoft entwickelt und ist eine objektorientiere Programmiersprache wie beispielsweise Java. Sie bietet daher auch verschiedenste bekannte Sprachkonstrukte wie Klassen, Vererbungen und Interfaces. Typescript ist jedoch eine Obermenge der 1996 entwickelten Sprache JavaScript. Im Vergleich zu dieser hat Typescript einige Vorteile, vor allem aber die starke Typisierung von Variablen, welche JavaScript nicht berücksichtigt. Dies führt zu deutlich weniger Fehlern und gestaltet das Verstehen des Codes signifikant einfacher. Letztendlich übersetzt der Typescript Compiler aber dennoch den Typescript Code in JavaScript Code, so ist jeder JavaScript Code auch automatisch korrekter Typescript Code.

Die wohl größten Vorteile von Typescript in Kombination mit Angular sind Modularität, Konsistenz sowie das zur Verfügung stehen verschiedenster Sprachkonstrukte. Angular wurde, wie im vorherigen Kapitel schon erwähnt, mit einer Komponentenstruktur entwickelt, welche einer programmierbaren Hierarchie entspricht. Diese Modularität ist mit Typescript sehr gut umzusetzen und führt zu einer klaren Struktur und einer guten Verständlichkeit des Programmcodes. Dank eben dieser Kombination zweier Technologien ist es auch mögliche eine hohe Konsistenz im Softwarecode zu erzielen, da eine Angular Komponente in Typescript immer den gleichen Aufbau haben muss.[[8]](#footnote-8)8)

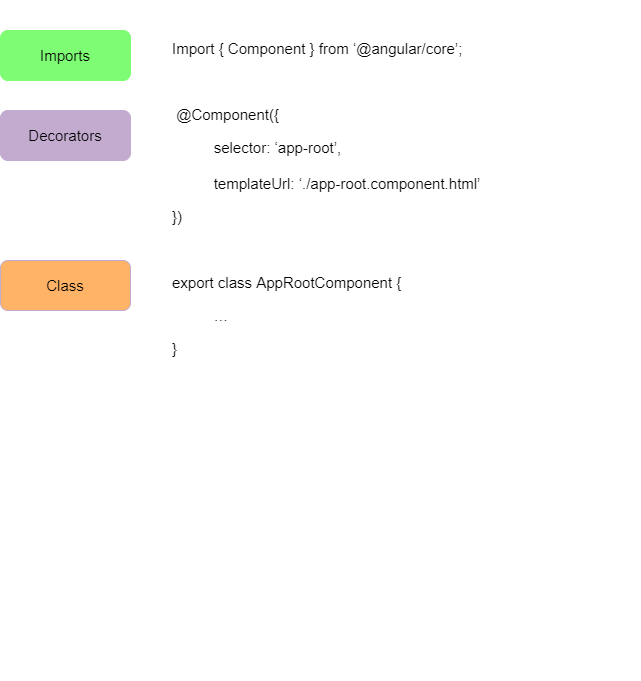


Abbildung 7: Aufbau einer Angular Komponente



# 4. Bereitstellen eines zentralen Repositorys

Es gilt ein neues Repository in dem bevorzugten Version Control System GitHub, oder auch Source Code Verwaltung, einzurichten. Es stehen zur Auswahl das Erstellen eines neuen Repos lokal auf dem eigenen Endgerät, welches man dann in GitHub pushen muss, oder eben das Erstellen eines Repos in GitHub direkt, welches anschließend auf das Endgerät zu pullen ist. In diesem Fall wird Variante zwei präferiert und zusätzlich soll ein SSH-Key generiert werden, um sich nicht bei jedem push oder pull Authentifizieren zu müssen. Das zu nutzende Betriebssystem ist Linux beziehungsweise Ubuntu 20.04, weshalb die meisten Arbeitsschritte im Terminal stattfinden.

Zu allererst muss das Repository in GitHub online erstellt werden, um auf das lokale Endgerät geklont zu werden. Hierzu bietet Git die Funktion Create new repository. Dort kann man anschließend einen Namen eintragen und die Sichtbarkeit von public auf private abändern, wenn gewünscht. Nun könnte man schon das bereits existierende Git Repository klonen und damit arbeiten, man müsste sich aber wie zuvor erwähnt bei jedem pull oder push authentifizieren. Dies wäre ein klonen über die HTTPS-Adresse des Repos, präferiert wird aber das klonen über das SSH Netzwerkprotokoll, wozu erst ein SSH Key eingerichtet werden muss. Dieser trägt dann zur Verknüpfung des GitHub Kontos mit dem Nutzer des Endgerätes bei. Durch den Befehl studienarbeit ~ $ ssh -keygen -t ed25519 -C ‚email@studienarbeit.com‘ wird dieser generiert und durch eingeben eines Passwortes in /home/USER/.ssh/id\_ed25519 gespeichert. In dem Ordner .ssh befinden sich dann mehrere Dateien, interessant für dieses Vorhaben ist aber lediglich id\_ed25519.pub. In dieser Datei befindet sich nämlich der öffentliche SSH-Key welcher nun in GitHub unter den Settings, SSH and GPG keys eingetragen werden muss. Nun ist der Nutzer des Endgerätes mit dem Nutzer von GitHub verknüpft und das Repo kann durch den Befehl studienarbeit ~ $ git clone git@github.com:USER/DHC.git geklont werden. Somit wurde das zentrale Repository erfolgreich erstellt.

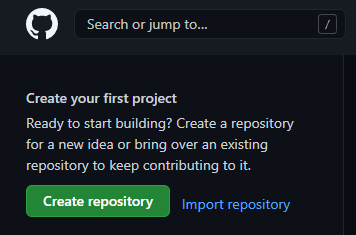


Abbildung 8: GitHub Create Repo

# 5. Anpassung des Build-Workflows

Das Unternehmen DaimlerTSS benutzt größtenteils zum Bauen ihrer Projekte das Build-System Bazel, welches 2015 von Google veröffentlicht wurde. Bazel ist ein Open-Source Tool welches zum automatischen Bauen von Softwarecode sowie zum automatisierten Testen dieses genutzt werden kann.[[9]](#footnote-9)9) Bekannte Firmen welche Bazel benutzen sind beispiesweise Adobe, Braintree, Etsy und Google.[[10]](#footnote-10)10)

Da Bazel auch für das automatisierte Bauen und Testen dieser drei Userinterfaces zuständig ist, muss der Build-Workflow angepasst werden, sodass Bazel beim Bauen das neue Repository mit den zentralisierten Komponenten auch findet und einbezieht. Dies geschieht in der sogenannten WORKSPACE Datei in einem jeweiligen Userinterface Ordner durch folgenden Code.

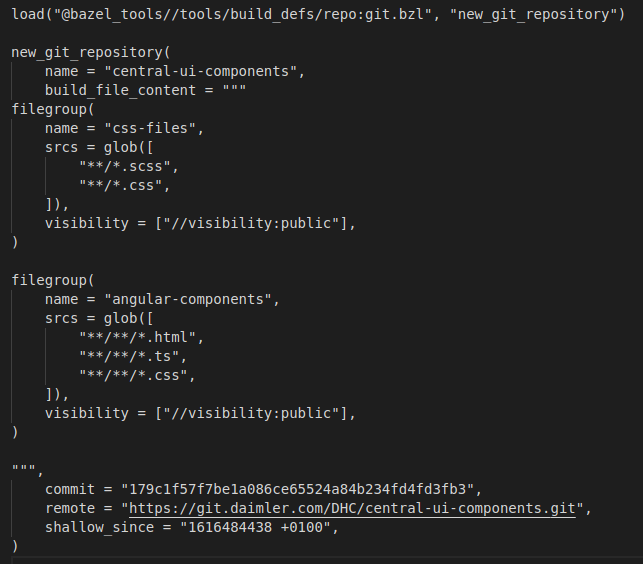


Abbildung 9: Anpassung des Build-Workflows

Im grünen Bereich wird durch new\_git\_repository( Bazel gesagt, dass es sich hierbei um ein neues Git Repository handelt, welches beim Bau berücksichtigt werden soll. Direkt dort drunter wird der Name des Repos angegeben und unter filegroup werden die zu zentralisierenden CSS-Styles und eine .scss Datei aufgeführt, welche im späteren Verlauf für Angular-Material Themes relevant ist. Der Abschnitt unter srcs = glob([ sagt schlicht weg aus, alle CSS- und SCSS-Dateien sollen berücksichtigt werden, egal in welchem Unterordner diese auch liegen. Nun muss was bereits für die Styles vorgenommen wurde, auch bei den Angular Komponenten Anwendung finden, dies geschieht im blauen Bereich. Der Filegroup wird ein Name zugeordnet und alle HTML, Typescript und CSS-Dateien sollen berücksichtigt werden. Wichtig ist es noch, den Commit-Hash des aktuellsten Stands des neuen Repositorys mitzugeben, sowie de Adresse des Repositorys, damit Bazel dieses findet und beim Bauen der gewünschte Stand verwendet wird.

Dies muss in jedem Userinterface separat gemacht werden und der Commit-Hash des neusten Standes muss bei einem neuen Release immer wieder händisch aktualisiert werden.

# 6. Kategorisieren einzelner Komponenten

## 6.1 Atome

Um die Zentralisierung erfolgreich anzugehen, müssen die zu zentralisierten Komponenten der Userinterfaces erst kategorisiert werden. Hierdurch wird garantiert, dass keine Komponente zentralisiert wird, welche dennoch Abhängigkeiten auf eine nicht zentralisierte hat. Dies kann zu Fehlern führen und der Sinn der Zentralisierung wäre verfehlt. Diese Kategorisierung erfolgt mithilfe des Atomic Design Pattern.

Zuallererst müssen alle Atome, welche in den drei Userinterfaces identisch sind, gefunden werden. Dies sind ausschließlich Stylings und befinden sich in den jeweiligen CSS-Dateien sich stark ähnelnder Angular Komponenten. Ein hierbei oft entdecktes Atom ist der a:hover, welcher benutzt wird wenn man über einen Link hovert.

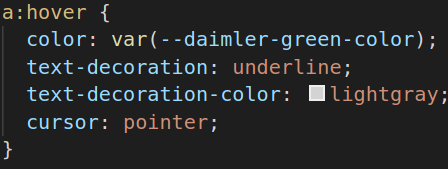


Abbildung 10: a:hover

Aber nicht nur der Hover-Effekt ist ein Styling für einen Link, welcher vermehrt in den drei Userinterfaces vorkommt, sondern auch viele andere, wie beispielsweise a:visited.

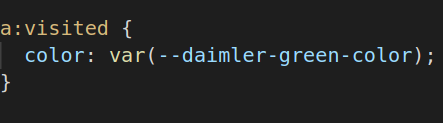


Abbildung 11: a:visited

Aber nicht nur Link Stylings in CSS sind zu zentralisierende Atome. Auch generelle Stylings eines Userinterfaces wie ein header-label in verschiedenen Ausführungen (dhc-page-header-label, dhc-small-page-header-label) und Stylings verschiedener Labels (dhc-small-label, dhc-small-bold-label) sind häufig vorkommende, meist identische und zu zentralisierende Komponenten.

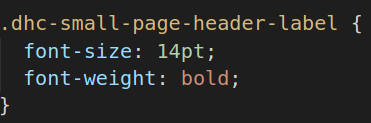


Abbildung 12: dhc-small-page-header-label

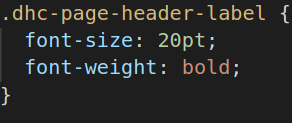


Abbildung 13: dhc-page-header-label

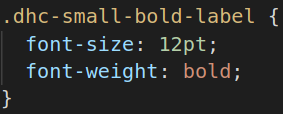


Abbildung 14: dhc-small-bold-label

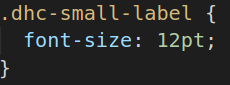


Abbildung 15: dhc-small-label

Auch ein sogennantes Loadingspinner.gif und sein CSS-Styling werden in den Userinterfaces sehr häufig verwendet, falls ein Inhalt nicht direkt geladen werden kann. Der Container in verschiedenen Ausführungen hierfür ist zur Beschränkung des Loadingspinners auf einen speziellen Bereich von Nöten. Auch diese zwei Komponenten sind zu zentralisierende Atome.

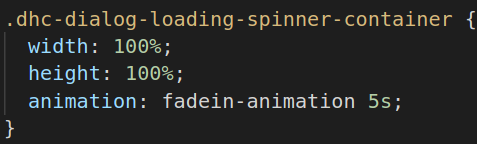


Abbildung 16: dhc-dialog-loading-spinner-container

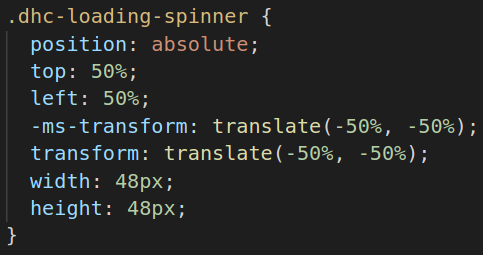


Abbildung 17: dhc-loading-spinner

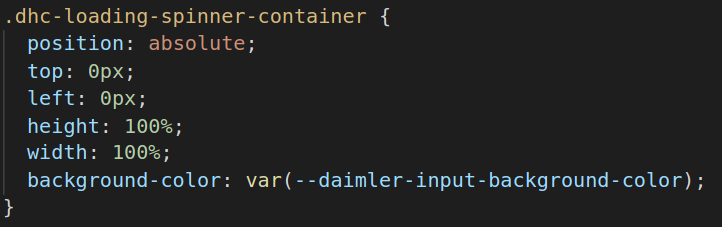


Abbildung 18: dhc- loading-spinner-container

Dies sind nur wenige Beispiele der zu zentralisierenden CSS-Stylings, welche als Atome kategorisiert werden. Aber schon bei diesen wird klar, eine Zentralisierung ist unausweichlich, denn nur schon in den CSS-Dateien befinden sich unzählige Codeduplizierungen von einem extremen Umfang. Aber auch die Userexperience wird dadurch verbessert, denn in den CSS-Stylings finden sich immer wieder kleine, aber signifikante Unterschiede. Mal ist das gleiche Label größer, hat eine andere Schriftgröße oder ein anderes Margin. Mit der Zentralisierung werden diese kleinen Unterscheide behoben.

## 6.2 Moleküle

Moleküle hingegen sind wie zuvor im Kapitel Atomic Design Pattern erklärt, ein Zusammenschluss aus mindestens zwei Atomen, welche in der Kombination einen Sinn ergeben. Auch für Moleküle finden sich Komponenten welche zu zentralisieren sind. Die Anzahl an Molekülen ist aber deutlich geringer als die der Atome. Die Angular-Komponenten Info-Dialog, License-Dialog sowie die Top-Bar werden hier aufgelistet.

### 6.2.1 Info-Dialog Komponente

Die Info-Dialog Komponente ist eine Angular-Komponente welche einen Pop-Up Dialog darstellt. In Abbildung 18 findet diese Komponente Anwendung als Help and support Pop-Up. Der Aufruf erfolgt über einen Need Help Button.

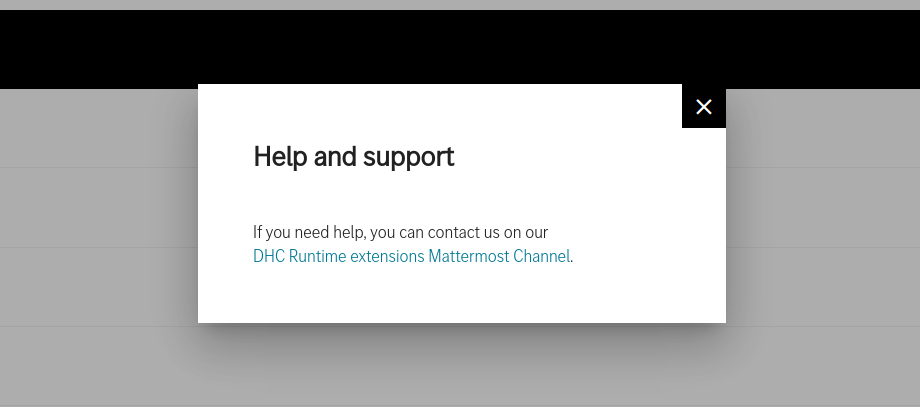


Abbildung 19: Info-Dialog Komponente

In diesem Beispiel findet die Info-Dialog Komponente Anwendung als Help and support Pop-Up. Wird der Aufruf der Komponente getätigt, so wird diese mit Inalt befüllt. In diesem Fall mit der needHelpDialogConfig().

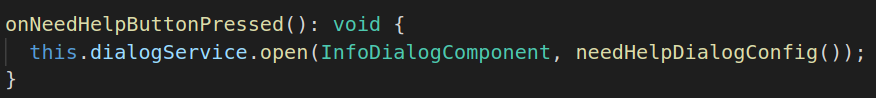


Abbildung 20: Aufruf der Info-Dialog Komponente

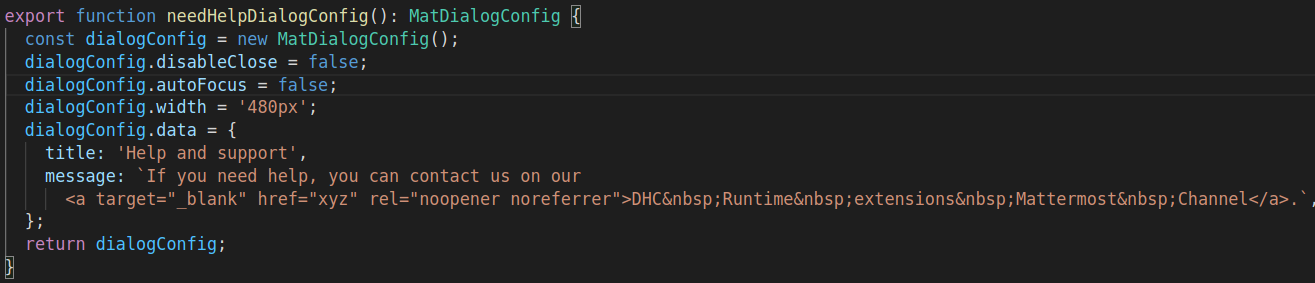


Abbildung 21: Inhalt der Info-Dialog Komponente

In Abbildung 21 hingegen wird der Info-Dialog in einem anderen Userinterface als Kontaktangabe verwendet. Somit wird auch hier schnell klar, die Zentralisierung ist dringendst von Nöten.

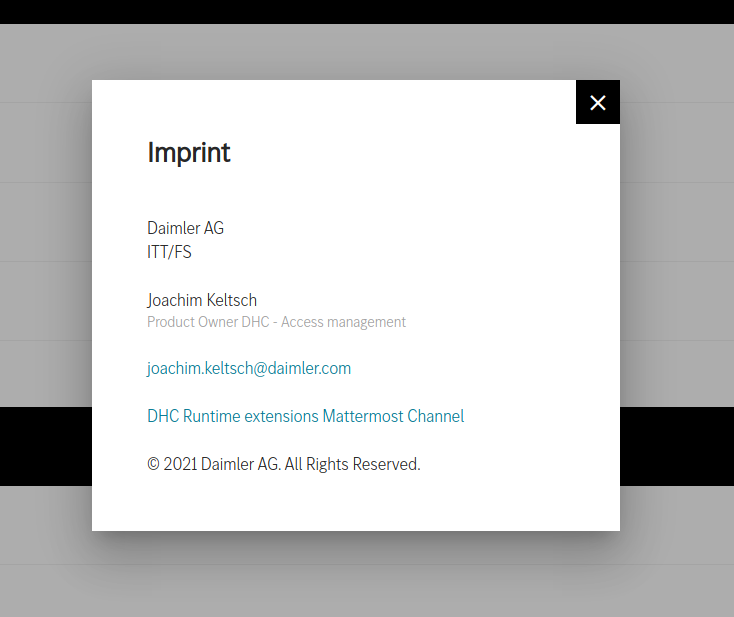


Abbildung 22: Info-Dialog als Kontaktangabe

### 6.2.2 Top-Bar Komponente

Die Anular Komponente Top-Bar beschreibt den Header jeder Website der jeweiligen Domain der drei Userinterfaces. Dieser wird nicht bei jeder neuen URL in einer Domain neu geladen, sondern wird in der zuvor schon erwähnten Datei app.component.html, mit der Syntax <app-top-bar></app-top-bar>, einmalig aufgerufen. Somit ist dieser statisch geladen. Die Top-Bar zeigt dem User mit dem Titel lediglich, in welchem Userinterface dieser sich befindet und bei einem Klick auf diesen wird er auf die Startseite jeder Domain weitergeleitet.



Abbildung 23: Top-Bar Komponente

### 6.2.3 License-Dialog Komponente

Die License-Dialog Komponente ist sehr ähnlich zur Info-Dialog Komponente. Sie ist ein Pop-Up welches einen License-Text wiedergibt. Der unterschied ist die Scrollbarkeit dieser Dialog Komponente, welche einige Änderungen im Sourcecode mit sich bringt. Außerdem wird diese Komponente mit einem vorgefertigtem Lizenztext gefüllt, welcher dort direkt eingebunden wird.

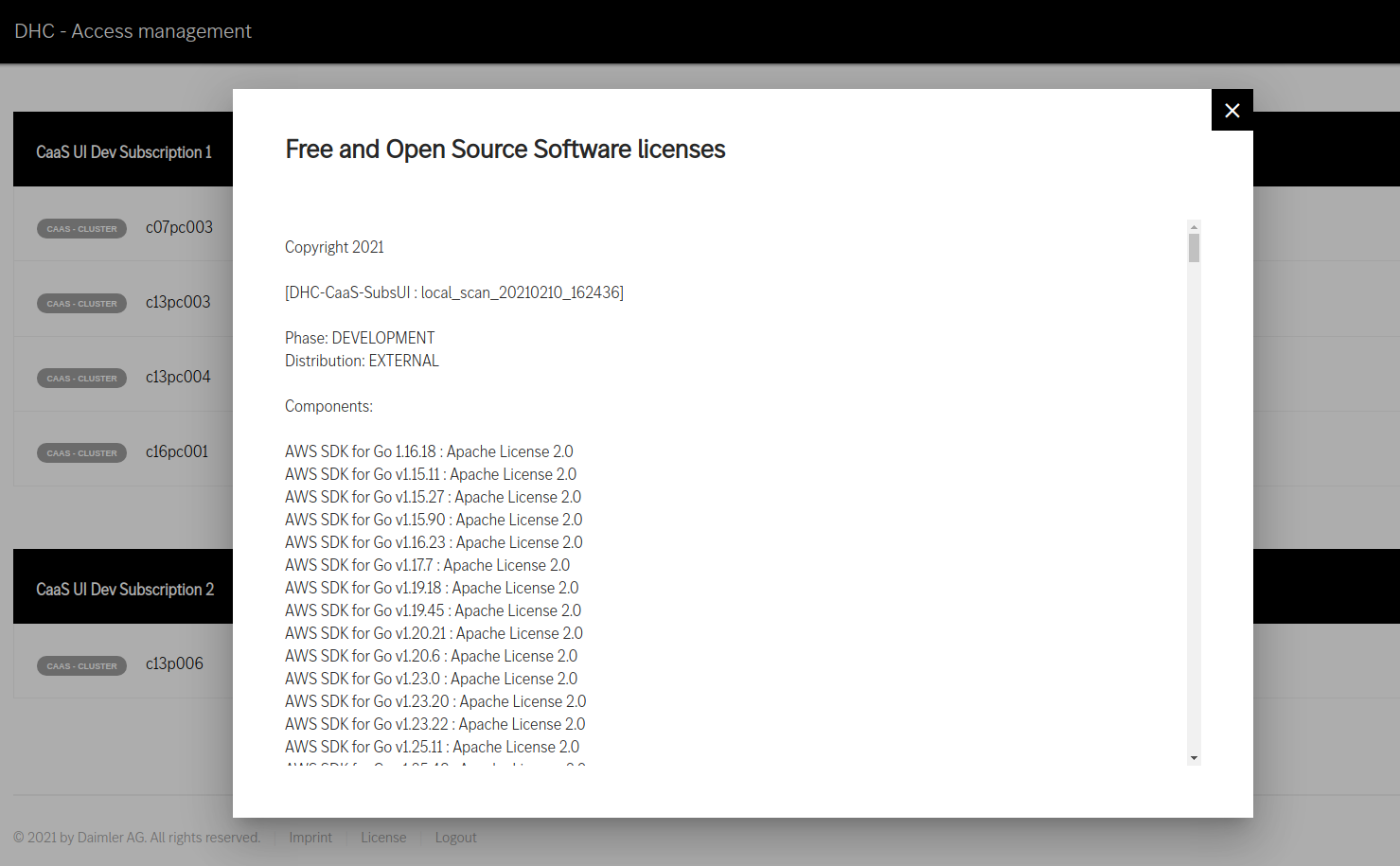


Abbildung 24: License-Dialog Komponente

## 6.3 Organismen

### 6.3.1 Copyright-Footer Komponente

Der einzige zu zentralisierende Organismus ist die Copyright-Footer Komponente, welche in Abbildung 23 gezeigt wird. Auf den ersten Blick sieht diese Komponente nicht komplex aus, ist es aber dennoch. Mit dem Klick auf Logout wird eine neue Seite aufgerufen, da dort noch eine Logout Möglichkeit implementiert werden soll. Beim Klick auf Imprint soll der Info-Dialog als Kontaktangabe genutzt werden und bei Auswahl des License Links wird die License-Dialog Komponente aufgerufen, in welche der License-Text wiedergegeben wird. Somit nutzt die Copyright-Footer Komponente andere Atome und Moleküle und ist somit ein Organismus.

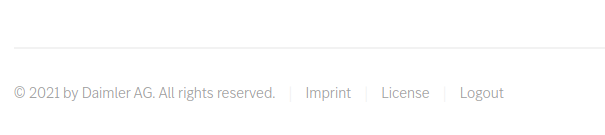


Abbildung 25: Copyright-Footer Komponente

## 6.4 Pages

### 6.4.1 Page-Not-Found Komponente

Im Kapitel Atomic Design Pattern wurden Pages als die komplexeste Einheit des Patterns deklariert, dies ist aber nicht immer der Fall. Die Page-Not-Found Komponente ist auch eine Page, welche aber sehr simpel gestaltet ist. Wird ein Verzeichnispfad beziehungsweise in Angular ein routingTarget, unter der vorgegeben Domain nicht gefunden, so wird per Default die Page-Not-Found Komponente aufgerufen. Diese Festlegung basiert in der schon zuvor erwähnten Datei app.module.ts.

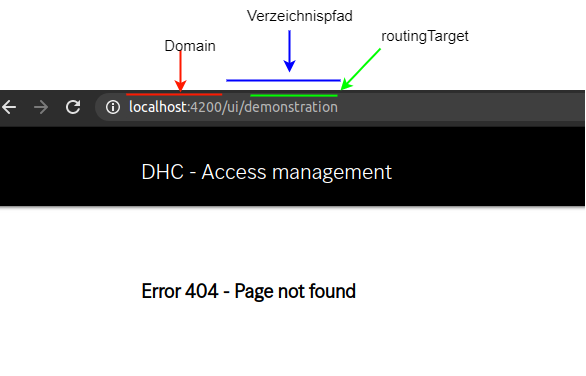


Abbildung 26: Page-Not-Found Komponente

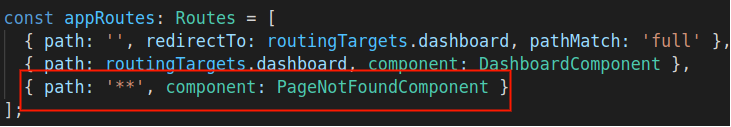


Abbildung 27: Page-Not-Found als Default in app.module.ts

# 7. Zentralisierung von Atomen

## 7.1 CSS-Stylings

CSS-Style haben keine Abhängigkeiten zu Molekülen oder ähnlichem und sind somit schnell und einfach zentralisiert. Im neuen Git Repository wird eine Datei names dhc-common-classes.css angelegt, in welche alle zu zentralisierenden CSS-Stylings eingefügt werden müssen. Wie in der Kategorisierung von Komponenten schon erwähnt, haben sich stark ähnelnde CSS-Stylings meist minimal Unterschiede wie ein Margin, welches sich um 2 Pixel unterscheidet oder eine kaum erkennbar andere Schriftfarbe. Diese Elemente werden nun zusammengefasst und das beste aus beiden wird übernommen. Außerdem werden verschiedene Benennungen behoben. Zum Beispiel werden am-loading-spinner und xy-loading-spinner zu dhc-loading-spinner, da ein Element nun nicht mehr nur in einer Abteilung separat, sondern in mehreren benutzt wird. Dieser kau signifikante Unterscheid in der Benennung hat aber zur Folge, dass jeder Aufruf eines solchen CSS-Elements nun abgeändert werden muss und dies in drei Userinterfaces.

## 7.2 Zentralisierung von Angular-Material Themes

Angular Material ist ein Frontend-Framework für vorgefertigte Komponenten oder Themes, sozusagen eine Bibliothek für Benutzeroberflächenkomponenten. [[11]](#footnote-11)9)Dort finden sich Komponenten wie Autocomplete, Buttons, Checkbox und viele mehr.[[12]](#footnote-12)10)

Die drei zu bearbeitenden Userinterfaces benutzen jeweils viele Angular-Material Themes, welche aber dennoch minimal abgeändert werden. Auch diese Styles gilt es in dem Zentralen Git Repository zu zentralisieren. Hierzu wird dort eine daimler-angular-material-themes.scss Datei angelegt und alle genutzten Angular-Material Themes und deren Änderungen werden dorthin verschoben. Beispiele für Angular-Material Themes sind folgende Abbildungen. Um Angular-Material Themes aber in diesem Repository nutzen zu können, muss eine package.json Datei eingefügt werden. Die package.json Dateien beinhalten alle zu importieren Abhängikeiten und deren Versionen. Ist eine package.json schon vorhanden, so muss nur den Befehl studienarbeit ~ /DHC/central-ui-components $ npm install oder auch studienarbeit ~ /DHC/central-ui-components $ yarn install durchgeführt werden, so werden alle Abhängigkeiten dort installiert. Ist die package.json Datei nicht vorhanden so kann der Befehl studienarbeit ~ /DHC/central-ui-components $ npm install - -save ausgeführt werden. Hierdurch wird auf Basis der bereits verwendeten Abhängigkeiten die package.json automatisch angelegt.



Abbildung 28: Angular-Material abgeänderte Buttons

Somit sind nun die Angular-Material Themes zentralisier- sowie in dem zentral bereitgestellten Repository nutzbar.

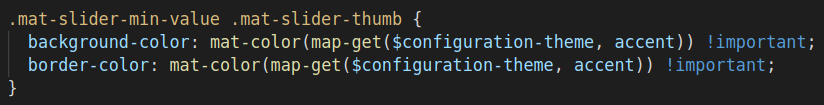


Abbildung 29: Anuglar-Material abgeänderter Slider

# 8. Zentralisierung von Molekülen

Da nun alle Atome zentralisiert sind kann mit den Molekülen weitergemacht werden. Würde man die Reihenfolge des Atomic-Design Patterns nicht beachten, so würde dies höchstwahrscheinlichest in mehreren Fehlermeldungen und schlechtem Programmierstyl resultieren

## 8.1 Info-Dialog Komponente

Um den Info-Dialog zu zentralisieren, muss der ganze info-dialog Ordner in das neuen Git Repository verschoben und lokal aus den Userinterfaces gelöscht werden. Um den Info-Dialog nun aufrufen zu können, werden in allen Dateien, welche den Info-Dialog benutzen, der Import so angepasst, dass dieser zur jetzigen Position der Komponente zugehörig ist. Der Komponente soll wie zuvor im Kapitel der Kategorisierung erwähnt, ein Parameter in Form eines MAT\_DIALOG\_DATA mitgeben werden, sodass diese Komponente oft wiederverwendbar ist und verschiedenste Texte in ihr angezeigt werden können. Dies geschieht nun mithilfe folgendes Codes.

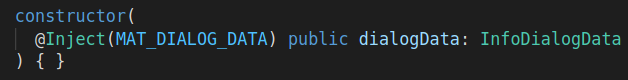


Abbildung 30: @Inject im Info-Dialog

@Inject ist eine sogenannte Dependency Injection in Angular. Mithilfe dieser kann beim erstellen eines Obejktes der Klasse Info-Dialog-Component ein Parameter in Form eines MAT\_DIALOG\_DATA mitgegeben werden.[[13]](#footnote-13)11) Ein Beispiel für einen solchen wird im Kapitel der Kategorisierung aufgeführt.

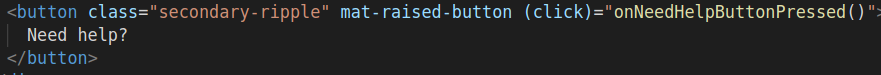


Abbildung 31: Klick auf Button "Need Help"

Durch einen Klick auf den Button Need Help wird nun folgende Funktion ausgeführt, welcher die MAT\_DIALOG\_DATA needHelpDialogConfig() mitgegeben wird. So wird der Info-Dialog mit dieser beladen und funktioniert als zentralisierte Komponente.

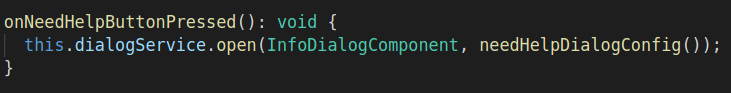


Abbildung 32: Aufruf des Info-Dialogs mit einer MAT\_DIALOG\_DATA

## 8.2 Top-Bar Komponente

Die zu zentralisierende Top-Bar enthält wie schon zuvor erwähnt, das Anzeigen des Namens des Userinterfaces, sowie die Funktion bei Klick auf den Namen die Weiterleitung auf die Startseite des jeweiligen Userinterfaces. Die Umsetzung hiervon wahr vor der Zentralisierung simpel, denn der Name sowie der Link waren Hardcoded in der Top-Bar. Dies soll sich nun ändern.

Auch der top-bar Ordner muss zuerst in das neue Repo kopiert und in den Userinterfaces gelöscht werden. Um den Titel der Top-Bar vom jeweiligen Aufruf mitgeben zu können, wird ein @Input in der top-bar.component.ts wie in folgendem Code verwendet. Außerdem wird für den Link der Startseite des Userinterfaces auch ein @Input bereitgestellt.[[14]](#footnote-14)12)

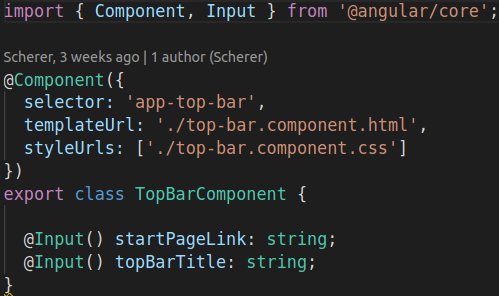


Abbildung 33: @Input in Top-Bar

Um nun die übergebenen Parameter an die Top-Bar nutzen zu können wird in der top-bar.component.html Datei der hardcodierte Startseitenlink sowie der Name des Userinterface durch diese Variablen ersetzt.

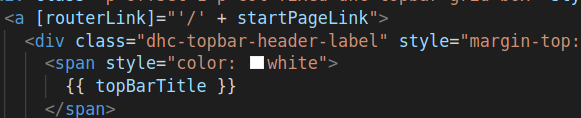


Abbildung 34: Ersetzten von hardcodiertem mit Variablen

Wie schon zuvor erwähnt wird die Top-bar Komponente statisch geladen und wird auf allen Seiten des Userinterfaces gleich angezeigt. Somit kann diese auch in der app.component.html Datei aufgerufen werden. Hier muss aber der Aufruf dieser Komponente abgeändert werden, sodass die gewünschten Parameter übergeben werden. Nun muss noch jeder Import der Top-Bar Komponente abgeändert werden und nun kann die Top-Bar Komponente in allen drei Userinterfaces über die app.component.html zentralisiert aufgerufen werden.

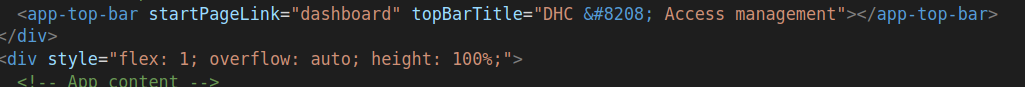


Abbildung 35: Neuer Aufruf der Top-Bar in app.component.html

## 8.3 License-Dialog Komponente

Die License-Dialog Komponente ist ähnlich zu zentralisieren wie die Info-Dialog Komponente, wird aber später von der Copyright-Footer Komponente und nicht mehr vom Dashboard direkt aufgerufen. Dies führt zu ein paar Änderungen dieser Komponente. Die License-Dialog Komponente bekommt ursprünglich vom jeweiligen Dashboard einen Pfad unter welchem ein Lizenztext liegt, einen Pfad für ein LoadingSpinner.gif sowie eine MAT\_DIALOG\_DATA zur Gestaltung des Textes in der Komponente, übergeben.

Da wie zuvor erwähnt die License-Dialog Komponente über den Footer aufgerufen werden muss, müsste man die zuvor genannten Parameter des Dashboards über den Copyright-Footer zum License-Dialog geben, was sehr umständlich ist. Nur mit @Injects und @Inputs ist dies leider nicht möglich, so muss zu anderen Maßnahmen gegriffen werden. MAT\_DIALOG\_DATA licensesDialogConfig() befinden sich anfangs nur Stylings zum License-Dialog. Diese licensesDialogConfig() wird eigentlcih vom Footer an die License-Dialog Komponente übergeben. Um die beiden anderen Parameter, den Pfad zum LoadingSpinner.gif und der zum Lizenztext, an diese Komponente übergeben zu können, werden diese in die licensesDialogConfig als eigenständige Parameter in der sogenannten dialogConfig.data geschrieben. So erhält die License-Dialog Komponente diese zum Zeitpunkt zur Erstellung des Objektes.

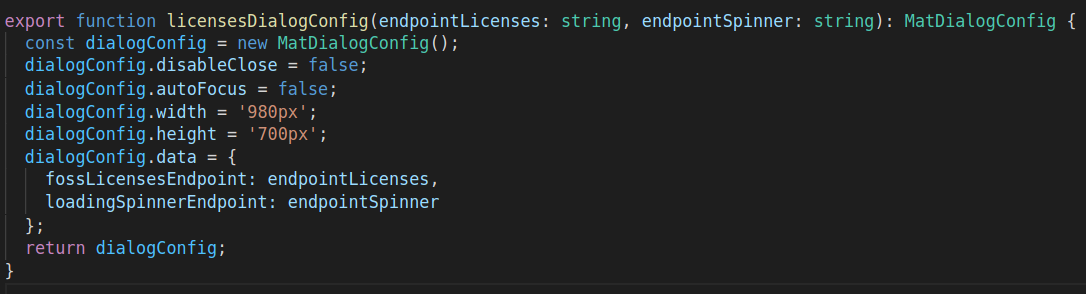


Abbildung 36: Hinzufügen der Pfade zur licenseDialogConfig()

Nun muss aber ein ngOnInit() zur Komponente hinzugefügt werden, welches bei Erzeugung des Objektes den LoadingSpinner.gif Pfad lädt, sowie den Pfad zum Lizenztext auf Richtigkeit überprüft und diesen anwendet. Somit kann die License-Dialog Komponente nun über den Copyright-Footer aufgerufen werden. Wie dies geschieht wird nun bei diesem gezeigt.

Literaturverzeichnis

Anke Lederer (2021): GitHub Eine praktische Einführung. Von den ersten Schritten bis zu eigenen GitHub Actions. 1. Aufl.: dpunkt.verlag GmbH.

Ashwin Sureshkumar (2019): @Inject and @Injectable. @Inject. Online verfügbar unter https://angular-2-training-book.rangle.io/di/angular2/inject\_and\_injectable, zuletzt aktualisiert am 2019, zuletzt geprüft am 10.05.2021.

Brad Frost (2016): Atomic Design.

Dan Wahlin (2019): Die 5 wesentlichen Vorteile von Angular und TypeScript. Hg. v. t2informatik GmbH. Berlin. Online verfügbar unter https://t2informatik.de/blog/softwareentwicklung/die-5-wesentlichen-vorteile-von-angular-und-typescript/, zuletzt geprüft am 22.04.2021.

Development Support (2021): Angular Material. Modernes Design für Angular Anwendungen. o.O. Online verfügbar unter https://dev-supp.de/programmierung/angular-net/angular-material#:~:text=Angular%20Material%20ist%20eine%20%22Benutzeroberfl%C3%A4chenkomponentenbibliothek,wurde%20und%20st%C3%A4ndig%20weiterentwickelt%20wird., zuletzt geprüft am 29.04.2021.

GitHub, Inc (2021): Where the world builds software. o.O. Online verfügbar unter https://github.com/, zuletzt geprüft am 22.04.2021.

Google: Angular Material offers a wide variety of UI components based on the Material Design specification. Online verfügbar unter https://material.angular.io/components/categories, zuletzt geprüft am 29.04.2021.

Google: Bazel overview. Online verfügbar unter https://docs.bazel.build/versions/4.0.0/bazel-overview.html, zuletzt geprüft am 10.05.2021.

Google: Sharing data between child and parent directives and components. Online verfügbar unter https://angular.io/guide/inputs-outputs.

Google: Who’s Using Bazel. Companies using Bazel. Online verfügbar unter https://bazel.build/users.html, zuletzt geprüft am 10.05.2021.

Malcher, Ferdinand; Hoppe, Johannes; Koppenhagen, Danny (2020): Angular. Grundlagen, fortgeschrittene Themen und Best Practices- inkl. RxJS, NgRx und PWA. 3. Aufl. Heidelberg: dpunkt.verlag GmbH.

Mark Boulton (2012): Structure First. Content Always. o.O. Online verfügbar unter https://markboulton.co.uk/journal/structure-first-content-always/, zuletzt aktualisiert am 06.02.2012, zuletzt geprüft am 20.04.2021.

1. 1) Vgl. Brad Frost (2016), S. 42. [↑](#footnote-ref-1)
2. 2) Vgl. Frost B. (2016), S. 44 f. [↑](#footnote-ref-2)
3. 3) Boulton M. (2012) [↑](#footnote-ref-3)
4. 4) Vgl. Frost B. (2016), S. 52 [↑](#footnote-ref-4)
5. 5) Lederer A. (2021), S. 2 [↑](#footnote-ref-5)
6. 6) GitHub (2021) [↑](#footnote-ref-6)
7. 7) Vgl. Malcher et al. 2020, S. 5ff [↑](#footnote-ref-7)
8. 8) Vgl. Wahlin D. (2019) [↑](#footnote-ref-8)
9. 9) Google. [↑](#footnote-ref-9)
10. 10) Google. [↑](#footnote-ref-10)
11. 9) Development Support (2021). [↑](#footnote-ref-11)
12. 10) Google. [↑](#footnote-ref-12)
13. 11) Sureshkumar A. (2019). [↑](#footnote-ref-13)
14. 12) Google. [↑](#footnote-ref-14)