# Warum Angular?

Wir haben uns aus zwei sehr einfachen Gründen für Angular als Framework für unser Frontend entschieden: Zum einen, weil ein Gruppenmitglied mit dem Framework bereits Erfahrung hatte, was uns erlaubt hat uns beim Lernen mehr auf Java EE und die Serverumgebung zu konzentrieren, zum anderen aufgrund von TypeScript. Anders als React, welches reines Standard-JavaScript benutzt, benutzt Angular TypeScript, was unglaublich viele Vorteile beim Programmieren bietet, zum einen wird der Code durch Typzuweisung (durch Interfaces, benutzeridentifizierte Typen, etc.) **deutlich** besser lesbar, der wohl größte Vorteil ist aber, dass man auf viele Fehler bereits beim kompilieren aufmerksam gemacht wird und diese nicht erst beim Ausführen bemerkt.

Zwar kann man React auch mit TypeScript kombinieren, da dieses zu JavaScript transpiliert wird, allerdings ist React, anders als Angular, nicht von vornherein darauf ausgelegt mit TypeScript zu arbeiten, was sehr viel Extraarbeit für die Programmierer bedeutet (unteranderem auch, weil React etwas zu viel Freiheit bei der Strukturierung des Codes lässt, anders als Angular, wo die Seite klar in verschiedene Module und Komponenten untereilt ist).

### Aber wieso nicht Flutter?

Beide Frameworks sind von Google und Flutter benutzt, anders als die anderen beiden Frameworks, eine ganz eigene Programmiersprache: Dart. Dart bietet genau wie TypeScript die Möglichkeit mit statischen Typen (wenn gewünscht) zu arbeiten. Allerdings ist Flutter primär darauf ausgelegt Apps und keine Websites zu programmieren. Es ist zwar möglich mit Flutter Websites zu erstellen, genauso ist es aber auch möglich mit Angular Apps zu erstellen, beide sind allerdings nicht primär für das jeweilig andere gedacht. Außerdem erlaubt Angular jede beliebige vorhandene JavaScript oder TypeScript Bibliothek mit einzubinden, da dieselbe Programmiersprache verwendet wird.

## Clarity als CSS-Framework

Als CSS-Framework haben wir zusätzlich Clarity benutzt, welches explizit für Angular entwickelt wurde und eine Reihe von sehr anpassbaren Komponenten und Modulen zur Verfügung stellt, sowie eine große Anzahl an Icons. Des Weiteren finden sich auch eine Menge Empfehlungen und Richtlinien im Umgang mit diesen Komponenten bezüglich des Designs auf der Website. Clarity ist mit dem vorgeschlagenen Material-UI-Design vergleichbar, beide bieten sehr ähnliche Elemente und Möglichkeiten, wir haben uns letztendlich aber für Clarity aus dem einfachen Grund, dass wir es ansprechender fanden und uns das Design besser gefallen hat.

# Struktur der Website und des Codes

## Allgemeine Struktur

Die Grundstruktur der Codes ist die standardmäßige von Angular generierte: Im Root-Ordner der Website finden sich diverse Konfigurations- und Informations-Dateien über das Projekt, der eigentliche Code befindet sich im *src* Unterordner.

Die Website an sich ist sehr stark Komponenten-basiert: Jede Seite besteht aus mindestens einer Komponente. Jede Komponente besteht aus mindestens 3 Dateien: Einem HTML-Template (*name.component.html*), einem Stylesheet für die Komponente (*name.component.scss*) und einer TypeScript Klasse, der eigentlichen Angular-Komponente (*name.component.ts*). Angular erlaubt allerdings Variablen, Funktionen usw. aus der TypeScript-Klasse direkt im HTML-Template einzubauen, mithilfe sogenannter „Directives“ lassen sich zum Beispiel bestimmte Elemente, Komponenten etc nur unter bestimmten Bedingungen anzeigen (mehr dazu in den Angular Docs). Jede Komponente definiert außerdem einen eigenen HTML-Tag, um diese anzusprechen, hiervon wird im Folgenden sehr viel Gebrauch gemacht.

Im src befindet sich die index.html Datei als Eingangspunkt für die Website, d.h. diese enthält nicht mehr als einige Konfigurations- und Meta-tags und verweist auf die eigentliche Website, dessen Code im „app“ Unterordner zu finden ist. Dazu noch weitere autogenerierte Dateien und eine weitere von uns erstellt Datei: „proxy.conf.json“, dazu später mehr.

Im „app“ Unterordner befinden sich 3 große Ordner: pages, ui und utils. Desweiteren finden sich in diesem Ordner das zentrale App-Modul, das Routingmodul und die zentrale App-Komponente. In der Datei *app.component.ts* finden sich einige Konfigurationen (zum Beispiel der Timer, der nach einer bestimmten Zeit den User wieder automatisch ausloggt), also alles, was global die ganze Website betrifft. Der HTML-Code verweist lediglich auf das Layout des UI-Moduls. Die letzte wichtige Datei hier ist das *app-routing.module.ts.* Dieses legt fest, welche Komponenten (welche im *pages*-Ordner zu finden sind), bei welchen URL-Pfaden angesprochen werden.

## Der UI Ordner

Der UI Ordner enthält die Komponenten für den „Rahmen“ der Website: Also den Header, die Sidebar und den Container für die anderen Komponenten, welche über den Router durch Angular automatisch angesteuert werden. Die Anordnung und das Wrapping der einzelnen Komponenten folgen dem von Clarity vorgesehenem Konzept, da es sich bei allen dreien um Clarity Komponenten handelt. Im Ordner finden sich nur das Ui-Modul und der Layout Unterordner, dieser enthält die Layout-Komponente, welche lediglich als Wrapper für die Header-, die Main- und die Sidebar-Komponenten dient (diese sind in ihren eigenen Unterordnern zu finden).

## Der Utils Ordner

Der src/app/utils Ordner beinhaltet alle eigentlichen Funktionalitäten der Website und hat noch drei weitere Unterordner: interfaces, services und pipes.

Zum einen findet sich in diesem Ordner der „AuthGuard“. Dieser Guard implementiert das „canActivate“-Interface von Angular, welches einen boolean zurück-liefert, ob der User die Seite betreten darf oder nicht (für welche Seiten das gilt wird im vorher beschriebenen *app-routing.module.ts* festgelegt). Konkret heißt das für unseren Fall: Der Guard liefert true zurück, wenn der User eingeloggt ist und somit weiter navigieren darf, ansonsten wird false zurückgeliefert und der User erhält eine Fehlermeldung.

Zum anderen die reuse.strategy.ts-Datei, in der sich nur eine einzige Klasse befindet, welche das Angular-„RouteReuseStrategy“ Interface implementiert, welches festlegt, ob, und wenn ja, welche Ansichten gespeichert werden sollen. Konkret auf unsere Seite bezogen heißt das: Wenn der User vom Dashboard weg (zu einer anderen Seite) navigiert und anschließend wieder zurück zum Dashboard, dann soll nicht alles noch einmal neu geladen und vom Server abgefragt werden (was mehrere Sekunden Verzögerung bedeuten würde), sondern noch vom letzten Besuch gespeichert werden soll. Falls der User allerdings die Daten aktualisiert haben will, findet er einen „Dashboard aktualisieren“-Button oben auf dem Dashboard.

Die Datei *validator.ts* enthält einen von uns erstellten „FormValidator“ für Angular, dieser kommt auf der Einstellungsseite zum Einsatz und prüft, ob zwei Benutzereingaben identisch sind. In unserem Fall wird dies angewendet, wenn der Benutzer sein Passwort ändern möchte. Mithilfe dieses Validators wird einfach nur überprüft, ob das neue Passwort und die Eingabe im „Neues Passwort bestätigen“-Feld übereinstimmen.

Die letzte wichtige Datei im utils-Ordner ist das Pipes-Modul, welches einfach nur erlaubt die von uns selber programmierten Pipes in anderen Modulen (also anderen Seiten) zu importieren. Pipes kommen in Angular HTML-Templates zum Einsatz, nehmen einen benutzerdefinierten Input und transformieren diesen. Unsere Pipes lassen sich im /pipes Unterordner finden und sind sehr einfach: Die reachablePipe nimmt einfach einen boolean als Wert und gibt wandelt true/false in ja/nein um, um diesen besser ausgeben zu können. (Der Name reachable kommt daher, dass diese Pipe beim „erreichbar“ Wert der Mitarbeiter zum Einsatz kommt und diesen transformiert.) Die zweite Pipe ist die employeeNamePipe, welche ein Mitarbeiter Objekt nimmt und den vollen Namen zurück gibt (da dieser in Mitarbeiter.vorname und Mitarbeiter.name aufgeteilt ist). Diese sind nicht kompliziert, helfen aber dennoch die HTML-Templates zu vereinfachen und etwas besser lesbar zu machen. Die dritte Pipe ist die listEmployeesPipe, welche einen Array von Mitarbeiter-Objekten nimmt und die Namen einfach aneinander reiht. Es kann außerdem eine Maximallänge übergeben werden: Falls diese erreicht wird, werden die restlichen Namen mit … referenziert und nicht weiter aufgelistet. Diese Pipe ist besonders hilfreich in Tabellen, da diese nicht immer groß genug sind alle Mitarbeiter anzuzeigen. Die vierte Pipe ist die accessRightsPipe, welche die system-internen Rechteklassen in gut lesbare (deutsche) Bezeichnungen umwandelt (zum Beispiel headOfDepartment zu Gruppenleiter).

Im /interfaces Unterordner finden sich einfach alle TypeScript-Interfaces, welche auf der Website zum Einsatz kommen. Wo diese zum Einsatz kommen, lässt sich einfach am Namen ablesen, dieser Folgt dem Schema *seite.model.ts* (zum Beispiel *dashboard.model.ts*), dazu gibt es noch eine *default.model.ts*-Datei, welche alle Interfaces enthält, welche global zum Einsatz kommen (zum Beispiel Mitarbeiter und Arbeitsgruppen).

## Die Services

Der wichtigste Ordner ist aber der /services Unterodner, obwohl dieser nur zwei Services enthält: Den ApiService und den DataService. Ein Angular Service ist eine Klasse, von der nur eine einzige Instanz erzeugt wird, welche sich beim Zugriff alle Komponenten „teilen“, dies spielt insbesondere bei der Modellierung des DataServices eine Rolle

Der ApiService dient, wie der Name schon sagt, zur Kommunikation mit der API, mithilfe von Angulars eingebauten HttpModule. Ein wichtiger Unterschied ist aber, dass wir uns entschieden haben anstatt den standardmäßig von Angular verwendeten Observables für die meisten Anfragen lieber Promises zu benutzen. Dies geschah aus dem einfachen Grund, dass die Website oft drauf angewiesen ist, auf das Ergebnis der Anfrage zu warten und dann zu entscheiden, ob zum Beispiel eine Fehlermeldung oder eine Erfolg ausgegeben werden soll. Dies geht mit einem *.then*-Chaining einfach deutlich einfacher, als mit Observables, welche sich nicht wirklich eignen, falls die Anfrage verschiedene Typen haben könnte (zum Beispiel eine Fehlermeldung oder ein Mitarbeiter-Objekt). Außerdem erlauben Promises asynchrone Funktionen, was heißt, dass mehre Komponenten gleichzeitig Anfragen mithilfe des ApiService an den Server senden können. An den Stellen, wo allerdings Observables einfacher zu nutzen sind (zum Beispiel um einfach nur ein HTML-Template zu füllen, ohne dass eine Fehlermeldung zurückgegeben werden könnte), können einige Abfragen auch optional als Observable zurückgegeben werden.

Der zweite Service, der DataService interagiert überhaupt nicht mit der API, sondern dient ausschließlich dazu, Daten lokal zu speichern, diese zu verwalten und an verschiedene Komponenten zu übergeben. So speichert er zum Beispiel das aktuelle Authentifizierungstoken für die API, welches der API-Service jedes Mal benutzen muss, um eine Anfrage zu stellen. Ebenfalls verwaltet er das Objekt für den aktuell eingeloggten Benutzer, dieses wird sofort lokal, also im DataService, gespeichert, da es einfach nicht sinnvoll ist jedes Mal wenn eine Seite aufgerufen wird die selben Daten abzufragen – schließlich werden die Daten des aktuell eingeloggten Benutzers so gut wie überall verwendet und dargestellt. Der DataService dient außerdem auch als Vermittler zwischen den Komponenten und dem localStorage (also dem Speicher auf dem PC des Benutzers), wo zum Beispiel die Präferenzen des Benutzers beim Bedienen der Website (zum Beispiel nach wie viel Minuten das automatische Ausloggen erfolgen soll, etc.) gespeichert werden. Diese Daten werden aber nicht einfach zurückgegeben, sondern mithilfe von BehaviourSubjects, einem weiteren RxJS-Datentypen gespeichert. Dies erlaubt es einzelnen Komponenten den Werten zu *subscriben*, wenn ein Wert geändert wird, wird dies sofort an alle momentan aktiven Komponenten übertragen und diese können dementsprechend drauf reagieren. Um beim Beispiel automatisches Logout zu bleiben: Wenn der Benutzer dies ändert, wird dies Änderung aufgrund des BehaviourSubjects sofort von der zentralen App-Komponente bemerkt, woraufhin diese den Timer mit der neuen Dauer neustarten kann. Wäre dies nicht möglich, müsste der User sich erst aus- und anschließend wieder einloggen.

Aus Sicherheitsgründen haben wir uns aber dazu entschieden, das User Objekt und das Auth-Token nicht im sessionStorage zu speichern, sondern in einer Variablen des Services. Dies macht für die Website keinen Unterschied, da alle Komponenten sich ja nur eine einzige Instanz des Services teilen. Ein Zugriff auf den sessionStorage wäre aber auch über die Konsole durch einen Benutzer des Browsers möglich, welcher somit ein Authentifizierungstoken oder alle Daten eines Benutzers in Erfahrung bringen könnte. Des Weiteren garantiert dies auch die Löschung der Daten, wenn die Website unerwartet schließen sollte, der sessionStorage würde erst automatisch geleert werden, wenn der Browser geschlossen wird und nicht wenn die Website geschlossen wird, wenn der Service also beim Löschen des Tokens unterbrochen werden würde, könnte ein Angreifer es immer noch in Erfahrung bringen und benutzen, wenn die Seite aber geschlossen wird, wird die Instanz garantiert vernichtet und somit werden auch die Daten gelöscht.

## Der Pages-Ordner

Der src/app/pages-Ordner ist der eigentliche Ausgangspunkt für die Website, dort (und in diversen Unterordnern) befinden sich die einzelnen view-Komponenten, also das, was der Benutzer am Ende sieht. Im Ordner gibt es ein zentrales Pages-Modul, dieses Verwaltet aber nicht alle Seiten, einige größere Seiten (zum Beispiel das Dashboard oder die Einstellungen) haben noch eigene Submodule, da diese noch eigene Sub- und Subsubkomponenten besitzen, welche für die restlichen Komponenten nicht relevant sind. Diese Module exportieren dann jeweils nur das final zusammengebaute Ergebnis (also eine einzelne dynamische Komponente, welche zwar Zugang zu Ihren Subkomponenten hat, andere Module und Komponenten aber nicht) und werden vom Pages Modul importiert.

Die Komponenten, die sich Konkret im Pages-Modul finden sind:

* Die EmployeeStatsComponent, welche als URL-Parameter (konfiguriert im *app-routing.module.ts*) eine ID von einem/r Mitarbeiter\*in nimmt und schließlich die Daten zu dieser Person ausgibt. Diese Komponente wird über *url/employee/:id* angesteuert. Admins können hier auch die Daten des jeweiligen Users (teilweise) bearbeiten.
* Die GroupStatsComponent funktioniert genauso, wie die zuvor erläuterte EmployeeStatsComponent, anstatt einer Mitarbeiter-ID nimmt diese Komponente aber die ID einer Arbeitsgruppe. Diese Komponente wird über *url/group/:id* angesteuert.
* Die ViewAllEmployeesComponent, welche als Übersicht für Admins bzw. die Personalabteilung über alle Mitarbeiter dient. Sie wird über *ur/view-all* angesprochen. Auch hier können Admins die Daten des jeweiligen Users (teilweise) bearbeiten.
* Die CreditsComponent wird über *url/credits* angesteuert und dient als eine Art Infoseite und zeigt die Credits an (also wer die Seite programmiert hat, etc…)
* Die PageNotFoundComponent wird angesteuert, falls das RoutingModule die URL keiner anderen Komponente zuordnen kann und zeigt einen Fehler 404 an – die Seite wurde nicht gefunden.

Eine Sache, die hier im Code bereits auffällt ist, dass der Code um Daten von der API abzufragen immer sehr ähnlich ausfällt und immer auch Fehler überprüft. Das Prinzip hier ist immer gleich: Es wird überprüft, ob es ein Fehlerobjekt ist und wenn ja erhält der User einen Alarm. Zwar hätte man das direkt im ApiService machen können und somit Code sparen können, allerdings haben wir uns dazu entschieden, dass die Fehlerüberprüfung immer in den Komponenten erfolgt, da der User einen Output basierend auf der Antwort erhalten soll und der Service nur als Vermittler dient: Er fragt die Daten nur an und leitet sie weiter. Die Komponenten sind aber dafür zuständig, was der User eigentlich sieht: Sie bekommen lediglich die Daten und stellen diese dar, mehr nicht. Außerdem erlaubt die Überprüfung in den Komponenten auch noch, dass bestimmte Signale oder Daten, basierend auf der Antwort der API, an die anderen Komponenten gesendet wird: Dies kommt besonders auf dem Dashboard zum Einsatz.

## Das CustomFormsModule

Das erste Submodul ist das CustomFormsModule im /forms Unterordner, es dient zur Verwaltung aller unserer selbst-erstellten komplexeren Formulare: Dem Loginformular, dem Löschformular und dem Registrierungsformular.

Dieses Modul enthält nur vier Komponenten:

* Die RegisterComponent, welche das Registrierungsformular beinhaltet und über *url/register* angesprochen wird.
* Die LoginComponent, welche das Loginformular beinhaltet und über *url/login* angesprochen wird und auch als Eingangspunkt zur Seite dient.
* Die DeleteComponenent, welche ein Formular zum Löschen eines Users beinhaltet und über *url/delete* angesprochen wird
* Die ErrorComponent, welche durch Angular one-way-binding (= die Komponente bekommt nur Input und gibt keinen Output an die hierarchisch höhere Komponente, welche sie aufruft zurück – das wäre sog. two-way-binding) einen Fehler bekommt und diesen entsprechend darstellt.

An der ErrorComponent sieht man auch bereits, wieso wir uns entschieden haben dafür ein eigenes Modul anzulegen: Die ErrorComponent wird nicht aus dem Modul exportiert und somit auch nicht in das PagesModule importiert, sie soll also nicht über eine URL ansprechbar und aufrufbar sein. Deshalb gehört sie nicht ins PagesModule und ist Teil des CustomFormsModule.

## Das SettingsModule

Das SettingsModule dient dazu die Einstellungsseite (ansteuerbar über *url/settings* zu verwalten) und besteht ebenfalls aus drei Komponenten, von denen aber nur eine Exportiert wird: Die SettingsComponent. Diese dient als übergeordneter Container und zur Strukturierung der Einstellungsseite. In der zugehörigen TypeScript-Klasse findet sich kein wirklicher auszuführender Code, fast die ganze Klasse besteht aus Konstanten Objekten. Diese werden an die Subkomponenten mittels ebenfalls mittels Angulars one-way-binding übergeben, das bringt mehrere sehr gute Vorteile mit sich:

1. Die Komponenten lassen sich dynamisch erstellen und beliebig oft sehr einfach wiederverwenden: Wenn ich dieselbe Komponente zweimal benutzten will und lediglich der Text unterschiedlich sein soll, kann ich einfach zwei verschiedene Inhaltsobjekte erstellen und an zwei verschiedene Instanzen der Komponente übergeben und muss nichts an ihrer Funktionsweise oder dem HTML-Template ändern (was ich müsste, wenn der Text statisch wäre).
2. Dies macht Änderungen an Formulierungen und Design sehr viel einfacher, ebenso wird das debuggen durch das runterbrechen in möglichst kleine Stücke deutlich beschleunigt und vereinfacht
3. Ebenfalls wird Ändern des Textes während die Seite aktiv ist deutlich ändern, da lediglich das Ändern eines Items in der Klasse reicht, um den neuen Text anzuzeigen

Die eigentlichen Komponenten des Settings-Modul sind die Account-Komponente, welche es dem User ermöglicht Änderungen an seinem eigenen Account vorzunehmen, und die select-Komponente, welche die HTML-templates für die Select Menüs auf der Seite beinhaltet. Wie zuvor erwähnt, erhalten beide Komponenten ihren darzustellenden Inhalt via Input durch die Settings-Komponente, der Text wird dann innerhalb dieser Komponenten dargestellt und entsprechend verarbeitet. Ebenso findet sich in diesen Komponenten dann das Übermitteln der Daten an den API-Service und das verarbeiten der Antwort der API.

## Das DashboardModule

Der wohl größte Teil der Website befindet sich im sehr großen DashboardModule: Es enthält ganze 10 Komponenten und ist auch wieder in einzelne Unterordner gegliedert. Im root-Ordner des Dashboardmoduls befindet sich nur die schon, relative große, Dashboard-Komponente. Diese tut allerdings nicht viel mehr, als die Daten vom Server abzufragen und anschließend auszuwerten, ob der User diese sehen darf oder nicht – falls ja, werden diese dann an kleinere Subkomponenten übergeben.

Eine Sache, die in dieser Komponente aber einzigartig ist, ist das Verwenden von Angular-Animations, welche wir sonst in keiner anderen Komponente auf der gesamten Seite verwendet haben. Angular-Animations funktionieren sehr ähnlich wie CSS3-Animationen (es werden zum Beispiel auch CSS-Attribute verwendet) aber haben den großen Vorteil, dass diese auch auf „versteckte Komponenten“ angewendet werden können. Wenn eine Komponente nicht sichtbar ist, ist sie nicht im DOM vorhanden, was heißt, dass keine CSS3-Animationen auf diese angewendet werden können, sie existieren ja quasi noch nicht. Angular-Animations lassen sich aber auf Komponenten anwenden, welche noch versteckt sind, da die Komponente das Element ja trotzdem kennt. Die „inOutAnimation“ die wir hier verwenden, sorgt lediglich dafür ein Element sichtbar bzw. unsichtbar zu machen, d.h. auf dem Dashboard die einzelnen Bereiche ein- oder auszuklappen.

Eine wichtige Methode hier ist die Methode „reload“, diese wird nie von der Komponente selber aufgerufen, sondern ausschließlich durch Output von sub-Komponenten. Diese verfügen, wenn notwendig, über den Output „refresh“, welcher ein EventEmitter ist und einen boolean zurückgibt. Wenn eine der Subkomponenten also true an das Dashboard emittet, wird dieses neugeladen. Das ist vorallem dann sinnvoll, wenn der User Änderungen vornimmt (zum Beispiel eine neue Gruppe hinzufügt oder eine entfernt), dann aktualisiert sich das Dashboard automatisch und die Änderungen werden sofort angezeigt.

Die erste Subkomponente ist die EmployeeDatagridComponent, welche dazu dient eine Liste von Mitarbeitern in Tabellenform darzustellen. Allerdings brauch diese Komponente einen Input: Entweder einen Mitarbeiterarray, welcher dann sofort dargestellt wird oder eine GruppenID, von dieser werden dann die Daten mittels des ApiService abgefragt und anschließend dargestellt.

Eine weitere kleiner Komponente ist die HeaderComponent, welche den Dashboard-Header beinhaltet: Hier kann der User seinen Status ändern, ob er abwesend oder anwesend ist (falls er abwesend sein sollte auch noch den Grund der Abwesenheit) und seine Daten aufrufen. Nach Änderung der Daten kommt hier auch der refresh EventEmitter zum Einsatz, nach durchführen einer Anfrage emittet die Komponente true an das Dashboard, welches daraufhin aktualisiert wird.

Der größte Unterordner im Dashboard ist der „groups“-Ordner, welcher alle Komponenten zur Verwaltung der Arbeitsgruppen beinhaltet. Die GroupsComponent nimmt als Input einen Array von Arbeitsgruppen, welche dann in Tabellenform dargestellt werden. Falls der User einer der entsprechenden Rechteklasse hat, wird auch noch ein „Arbeitsgruppe hinzufügen“ Button angezeigt und es erscheint ein kleiner Mülleimer neben einem Gruppennamen, welcher es möglich macht die Arbeitsgruppe zu löschen. Auch diese Komponente verfügt über einen refresh EventEmitter, welcher allerdings nie von der Komponente selber getriggert wird sondern nur von einer der drei Unterkomponenten, welche sich in diesem Ordner befinden und anschließend an das Dashboard weitergegeben. Die drei Komponenten sind:

* Die AddGroupWizardComponent, welche einen Wizard zum Hinzufügen einer Gruppe ist und aus einem einfachen Fenster mit einem Formular besteht. Wenn der User den Vorgang abgeschlossen hat, wird true durch den refresh EventEmitter emittet und durch die GroupsComponent an das Dashboard weitergegeben.
* Die AddUserToGroupComponent, welche dazu dient einen Benutzer zu einer Gruppe hinzuzufügen (oder aus dieser zu entfernen, wenn angegeben). Auch diese besteht nur aus einem Popup mit einem Formular, was anschließend die Daten an den ApiService übergibt und dann true emittet.
* Die RemoveGroupComponent, welche ebenfalls aus einem Popup-Fenster mit einem Formular besteht und anschließend die Daten an den ApiService übergibt und true emittet. Das Formular hier dient aber, wie der Namen bereits sagt, dazu eine Gruppe zu löschen.

Auch wenn alle drei Komponenten zumindest in ihrem HTML zum Teil ähnlich sind, sind sie doch in unterschiedliche Komponenten zerlegt, da sie an sehr unterschiedlichen Stellen zum Einsatz kommen und sich die TypeScript-Klassen teilweise sehr stark unterscheiden. Man könnte diese zwar zusammenlegen, um Code zu sparen, das würde es aber nur unübersichtlicher machen, da man viel mehr Prüfungen bräuchte, welche Anfrage denn jetzt gerade versendet werden soll.

Der letzte Unterordner im Dashboard-Ordner ist der mobile-Ordner, welcher die mobilen Ansichten für einzelne Komponenten beinhaltet, da nicht alle (zum Beispiel die Tabellen) sehr gut auf einem Mobilgerät anzusehen sind. Die TypeScript-Klassen und Datenverarbeitung dieser ist aber sehr ähnlich zum „Desktoporiginal“, unterscheidet sich aber leicht- Im Detail sind diese Komponenten:

* Die AddGroupMobileComponent, welche ein alternatives Fenster zum Erstellen einer Gruppe zur Verfügung stellt, da der Wizard aus der AddGroupWizardComponent etwas zu breit für die Mobile Ansicht ist.
* Die PanelsComponent, welche das mobile Äquivalent der EmployeeDatagridComponent darstellt: Auf Mobilgeräten werden die Mitarbeiter aber nicht als Tabelle sonder halt als Panels dargestellt. Das heißt man sieht nur die Namen der Mitarbeiter und falls man mehr sehen möchte, klickt man einfach drauf und es öffnet sich ein Dropdown mit mehr Informationen.
* Die HeaderComponent, welche einfach die mobile Version der eigentlichen HeaderCOmponent ist, auch diese Arbeitet mit Panels, da vertikales Design einfach besser für Mobilgeräte geeignet ist

# Installation

Das Installieren und Inbetriebnahme der Website ist sehr einfach. Die einzige Voraussetzung ist, dass Node.js installiert ist (für Angular 9 ist Node.js v12+ empfohlen). Node.js können Sie hier (<https://nodejs.org/de/download/>) herunterladen. Der gesamte Installation- und Startvorgang erfolgt über die Konsole. Im Folgenden werde ich die Befehle, welche man auf einem Windows PC verwendet müsste, angeben.

Sobald Sie Node.js installiert haben, wechseln Sie vom Root Ordner des Sourcecodes in den /Frontend/ Unterordner mittels „cd Frontend/“. Anschließend installieren Sie zuerst Angular mittels npm (Node Package Manager), indem Sie „npm i @angular/cli@9.0.2 –g“. Dies installiert Angular global, sodass Sie auch die Angularbefehle in der Konsole nutzen können. Anschließend installieren Sie alle weiteren Bibliotheken, indem sie „npm i“ ausführen (alle Bibliotheken sind in der Datei „package.json“ zu finden). Wenn dieser Vorgang abgeschlossen ist, führen Sie „ng serve –prod“ aus. Die Production-Flag (--prod) darf auf keinen Fall weggelassen werde, da die Anwendung sonst im Testmodus gestartet wird und nicht mit dem Server kommuniziert, sondern lediglich Beispieldaten verwendet.

Wenn die Website fertig kompiliert ist, öffnen Sie <http://localhost:4200/> und Sie sehen die Website.

Alternativ lässt sich auch Angulars build feature verwenden. Anstatt „ng serve --prod", führen Sie dann „ng build --prod" aus. Dies bündelt die gesamte Website in wenige (sehr große Dateien), diese müssen *bla bla Nils sag mal worein*

Für den Build-Vorgang steht auch ein Dockerfile im Rootordner des Sourcecodes zu verfügung