



# Globalizer

Detailkonzpet

**Koller, Jonas**

`jonas.koller@gmx.ch`

**Wolfisberg, Donato**

`donato.wolfisberg@gmail.com`

18.10.2018





## 1 Einleitung

Dieses Dokument behandelt das Detailkonzept des Projekts Globalizer. Ziel und Zweck des Dokuments ist es, die fachlichen Anforderungen technisch zu umschreiben, damit diese später implementiert werden können. Zudem soll das Gesamtbild unserer Applikation klar gemacht werden und die Komponenten, sowie der Datenaustausch genau beschrieben werden. Es werden auch all unsere Technologieentscheide vermerkt und begründet. Zusammengefasst soll dieses Dokument also eine klare, wie auch detaillierte Übersicht zu den technischen Aspekten unseres Projekts sein.

## 2 Allgemeine Informationen

Hier folgt eine kurze Auflistung der Informationen zu diesem Dokument, dem Entwicklungsteam und dem aktuellen Stand.

### 2.1 Projektmitarbeiter

Name	Vorname	E-Mail	Funktion
Koller	Jonas	jonas.koller@gmx.ch	Projektleiter
Wolfisberg	Donato	donato.wolfisberg@gmail.com	Entwickler
Gian	Ott	gian_ott@sluz.ch	Prüfer
Manuel	Troxler	manuel_troxler@sluz.ch	Prüfer

### 2.2 Änderungskontrolle

Version	Datum	Ausführende Stelle	Bemerkung
1	21.09.2018	Projektteam	Erste Version des Dokuments erstellt
2	23.09.2018	Projektteam	Gesamtüberblick erstellt
3	25.09.2018	Projektteam	Zielkatalog erstellt
4	27.09.2018	Projektteam	Abschliessende Arbeiten

### 2.3 Prüfung

Version	Datum	Ausführende Stelle
4	27.09.2018	Gian Ott

# Inhalt

<b>1</b>	<b>Einleitung</b>	<b>I</b>
<b>2</b>	<b>Allgemeine Informationen</b>	<b>II</b>
2.1	Projektmitarbeiter . . . . .	II
2.2	�nderungskontrolle . . . . .	II
2.3	Pr�fung . . . . .	II
<b>3</b>	<b>Systemkomponenten</b>	<b>1</b>
3.1	Frontend . . . . .	1
3.2	Backend . . . . .	1
3.3	Datenbank . . . . .	1
3.4	Zonen-�bersicht . . . . .	2
<b>4</b>	<b>Design-Pattern</b>	<b>2</b>
4.1	Allgemeine Architektur-Pattern . . . . .	2
4.2	Codestyle und Codequalit�t . . . . .	3
4.3	Architektur Backend . . . . .	3
4.3.1	UML Flussdiagramm . . . . .	6
4.4	Architektur Frontend . . . . .	7
<b>5</b>	<b>Persistenz</b>	<b>7</b>
<b>6</b>	<b>Detaillierte Komponenten und Schnittstellen</b>	<b>7</b>
6.1	Backend . . . . .	7
6.2	Funktionale Anforderungen . . . . .	7
6.3	Schnittstellen . . . . .	8
6.4	Persistenz . . . . .	8
6.5	Frontend . . . . .	8
6.5.1	UML Flussdiagramm . . . . .	8
6.6	Funktionale Anforderungen . . . . .	9
6.7	Schnittstellen . . . . .	9
6.7.1	Android frame . . . . .	9
6.7.2	Desktop Chrome Login . . . . .	10
6.7.3	Desktop Chrome Chat . . . . .	11
6.8	Persistenz . . . . .	11

<b>7</b>	<b>Technologieentscheide</b>	<b>11</b>
7.1	Webclient . . . . .	12
7.2	Angular . . . . .	12
7.3	NodeJS . . . . .	12
7.4	Github . . . . .	12
7.5	LaTex . . . . .	13
<b>8</b>	<b>Continuous Integration und Continuous Testing</b>	<b>13</b>
8.1	CI mit CircleCI . . . . .	13
8.2	CT mit Cypress . . . . .	13
8.3	Code Review mit CodeFactor . . . . .	13
8.4	Darstellung CI & CT . . . . .	14

## 3 Systemkomponenten

Das Projekt verwendet eine klassische Client - Server Architektur. Somit ergeben sich bei uns drei Hauptkomponenten. Dies ist das Backend, das Frontend und die Datenbank. Wir werden folgend beschreiben, wie diese genau aufgebaut sind.

### 3.1 Frontend

In der Frontend-Komponente soll die Schnittstelle zwischen Endbenutzer und Backend-System implementiert werden. Wir werden dies mit einer Weboberfläche machen. Diese Komponente soll unabhängig vom Backend-System auf einem CDN gehostet werden können. Diese Massnahme verringert die Wartezeiten des Endnutzers deutlich.

### 3.2 Backend

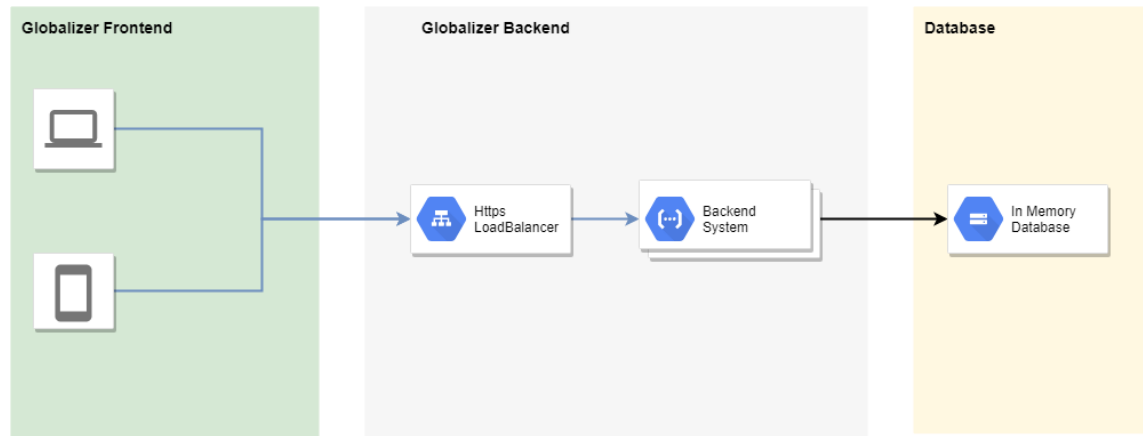
Unser Backend-System soll die Schnittstelle zwischen den Daten und dem Frontend abbilden. Hier werden die rohen Daten von der Datenbank aufbereitet, Berechnungen durchgeführt und die Security-Aspekte grösstenteils abgedeckt. Die Authentifizierung, Autorisierung und Überprüfung der übermittelten Daten soll hier stattfinden. Das Backend soll möglichst klein gehalten werden, damit der Wartungsaufwand minimal bleibt. Es soll so gebaut werden, dass es gut skalierbar verwendet werden kann. Aspekte wie Load-Balancing und Upscaling sollen vom Hoster übernommen werden.

### 3.3 Datenbank

Die Datenbank bildet den dritten Teil unseres Systems. Hier soll eine InMemory-Datenbank verwendet werden. Diese braucht weniger Leistung und sorgt für zusätzliche Geschwindigkeit beim System. Uns ist bewusst, dass die Datenbank ihren aktuellen Stand bei einem Systemabsturz verlieren kann. Da es sich jedoch um einen flüchtigen Chat handelt, welcher nicht zu 100% sicher persistiert werden muss, kann eine InMemory-DB verwendet werden.

### 3.4 Zonen-Übersicht

Auf der nachfolgenden Grafik kann unsere Architektur im groben ausgelesen werden. Das System ist in die drei "Komponenten-Zonen" unterteilt. Diese sollen wenn möglich unabhängig von den anderen deploybar sein. Dies ermöglicht es uns später auch, unsere Deployment-Zyklen zu vereinfachen. Im Backend-System ist zusätzlich auch der LoadBalancer eingezeichnet, welcher vom Hoster übernommen wird.



## 4 Design-Pattern

Da weder unser Backend, noch unser Frontend objektorientiert ist, können keine Klassendiagramme erstellt werden. OOP-Eigenschaften wie Vererbungen und Beziehungen werden bei diesem Projekt auch kaum benutzt. Aus diesem Grund verzichten wir auf diese Diagramme. Wir verwenden anstelle dessen aber ein UML-Aktivitätsdiagramm, um unsere Abläufe zu visualisieren. Wir verwenden in diesem Projekt keine Domain-Driven-Design, Test-Driven-Design oder ähnliches, da unser Projekt zu klein ist, damit dies den Aufwand wert wäre.





### 4.1 Allgemeine Architektur-Pattern

Wir wollen die Kommunikation zwischen Frontend und Backend möglichst dynamisch und asynchron halten, damit weder das Front- noch das Backend lange Wartezeiten aufweist oder "einfriert". Da die Kommunikation selbst aufgrund der Technologien nur schwer ganz asynchron umsetzbar ist, verwenden wir das in Angular häufig angewandte "pseudo Asynchronität". Die Anfragen an sich verlaufen synchron. Angular lässt aber zu, dass das Frontend währenddessen nicht blockiert wird und

arbeitet bis zur Antwort den restlichen Befehlsstack ab und h  rt auf Events. Im Backend l  uft das auch so. Somit blockieren weder das Back- noch das Frontend und der Nutzer hat ein vielfach besseres Erlebnis beim verwenden der Applikation. Sowohl verwenden auch an beiden Orten das "Observer"-Pattern. Dies wird durch Angular und NodeJS unterst  tzt, in dem wir die "rxjs"-Library verwenden. Durch die einbindung dieser vereinfacht sich das Behandeln von Events und das reaktive Programmieren.

## 4.2 Codestyle und Codequalit  t

Damit unser Projekt auch   ber lange Zeit verwendbar, Wartbar und Erweiterbar ist, werden wir einige Programmierprinzipien anwenden, welche sicherstellen, dass unser Code langfristig sauber bleibt. Dies sind folgende:

-  Cleancode-Prinzip
-  SOLID-Prinzip
-  Pair-Programming-Prinzip
-  Feature-Banches und Pull-Requests

Falls einer der Begriffe unbekannt sein sollte, kann eine Erkl  rung zu diesem im Internet gefunden werden.

Wir verwenden Github als Versionskontrolle (mehr dazu bei den Technologieentscheidungen). Damit wir die oben genannten Prinzipien einhalten, setzen wir Codefactory ein. Zudem verwenden wir auch TSLint, eine Clientseitige Library zur   berpr  fung vom Codestyle.

Desweiteren verwenden wir einen Continuous Build System und ein Continuous Testing System, welches aktiviert wird bei einem "Push" ins Github Repository. Dazu auch sp  ter mehr.

## 4.3 Architektur Backend

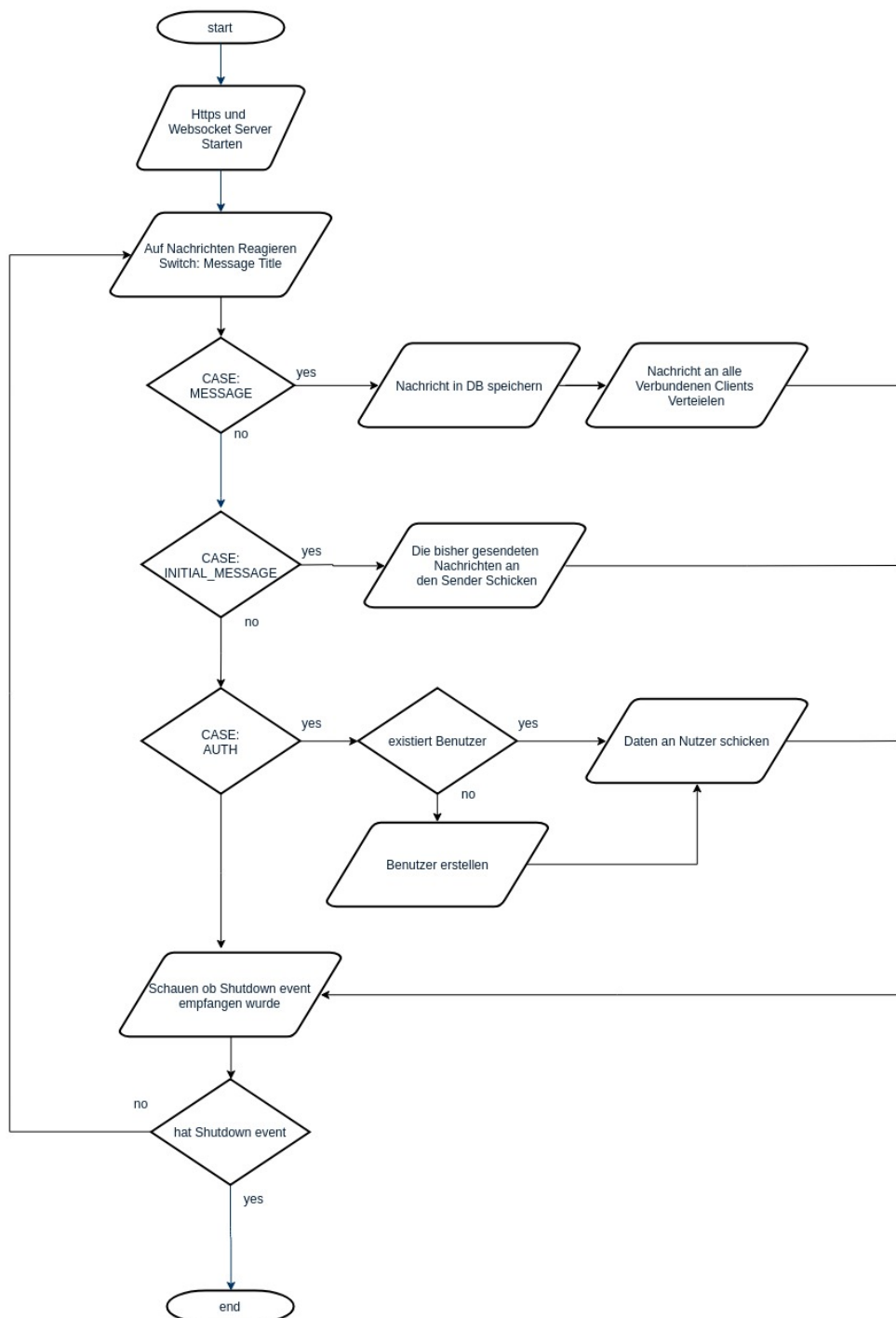
Das Backend wird in NodeJS erstellt. Es nutzt Websockets zur Kommunikation mit dem Frontend. Das WebSocket-Protokoll ist auf TCP basiert und erlaubt bidirektionale Verbindungen zwischen Server und Client. Das Backend ist offen f  r neue Verbindungen. Wenn sich ein neuer Benutzer anmeldet, registriert dieser sich beim Backend. Dieses h  rt danach auf Anfragen des Frontends. Im gegenzug sendet das Backend neue Nachrichten selbst an das Frontend. Durch dieses "Bidirectional



Message Pattern" können wir einiges an Bandbreite einsparen, da wir nicht immer wieder Fetch-Anfragen senden müssen um den Client auf dem aktuellen Stand zu behalten. Wir konnten somit das "Fetch"-Antipattern umgehen, da dieses für eine Chat-Applikation ungeeignet ist. In der folgenden Grafik kann ausgelesen werden, wie der Ablauf bei Anfragen an den Server aussehen.



### 4.3.1 UML Flussdiagramm



## 4.4 Architektur Frontend

Unser Frontend wird mit Angular (Version 7) implementiert. Wieso genau wir uns f  r diese Technologie entschieden haben wird sp  ter erl  utert. Dazu wird das Google Angular Material verwendet, damit wir dank dieser Library nicht alle UI-Komponenten selbst erstellen und designen m  ssen, sondern auf diese UI-Library zur  ckgreifen k  nnen (diese Library ist   hnlich wie Twitters Bootstrap). Das Frontend sendet auch via Websocket-Protokoll Nachrichten an das Backend.

## 5 Persistenz

Unsere Applikation verwendet wie bereits oben beschrieben eine InMemory Datenbank. Diese wird eine NoSQL Datenbank sein (NoSQL im Sinne von kein SQL verwendend und nicht darauf basierend). Sie basiert auf einem JSON-Datenmodell. Dies da wir in JavaScript programmieren werden und die Daten verwenden k  nnen, ohne diese zuerst parsen zu m  ssen. Die InMemory-DB wird durch Vanilla-JavaScript erm  glicht und braucht keine weitere Library. Ein konkretes Datenmodell gibt es nicht aufgrund von dem Fakt, dass wir dynamisch Daten speichern und entfernen und das genaue Datenmodell so entsteht.

## 6 Detaillierte Komponenten und Schnittstellen




Folgend werden alle Komponenten genau beschrieben und deren Schnittstellen definiert.

### 6.1 Backend

Die Backendkomponente bildet das "Hirn" unserer Applikation. Hier werden sowohl alle wichtigen Daten gespeichert (in der InMemory-DB), als auch alle wichtigen Impulse verwendet (Websockets). Ziel des Backends ist es eine m  glichst hohe Flexibilit  t und Wartbarkeit mit einer m  glichst hohen Performance zu kombinieren. Dies wird durch die sehr junge Backend-Sprache NodeJS erm  glicht.

### 6.2 Funktionale Anforderungen

Das Backend ist betroffen von 3 obligatorischen funktionalen Anforderungen. Diese sind:

-  USER\_STANDARD\_LOGIN
-  NACHRICHT\_SENDEN
-  NACHRICHT\_EMPFANGEN

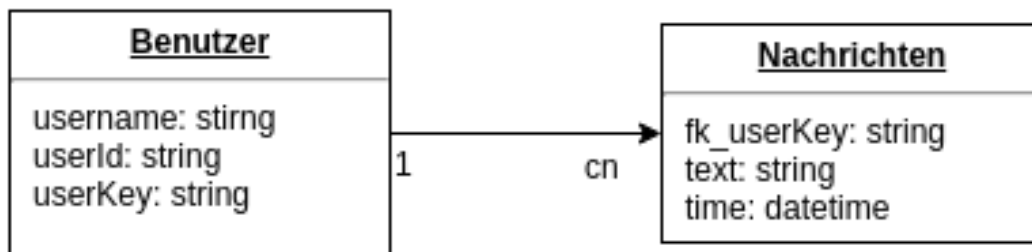
Was diese Anforderungen konkret bedeuten kann im Dokument "Anforderungsspezifikationen" nachgelesen werden. Dort sind diese inklusiv Use-Case-Diagramm dargestellt.

## 6.3 Schnittstellen

//TODO: Schnittstellen beschreiben

## 6.4 Persistenz

Das Backend   bernimmt den Hauptteil der Persistenz der ganzen Applikation. Hier werden alle Nachrichten und Benutzer in die InMemory-DB gespeichert und der Zustand der ganzen Applikation. Die Entit  ten sehen wie gefolgt aus:



## 6.5 Frontend




Die Frontendkomponente bildet die Schnittstelle zwischen Endbenutzer und System. Hier werden die Daten aus dem Backend grafisch dargestellt. Wichtig an dieser Komponente ist, dass das Aussehen dieser modern und stylich ist, damit der Nutzer ein tolles Erlebnis auf der Seite hat. Auch Geschwindigkeit ist hier ein wichtiges Thema, weshalb Angular 7 eine sehr gut passende Technologie ist. Die Funktionsweise des Frontends kann aus diesen UML-Diagramm ausgelesen werden.

### 6.5.1 UML Flussdiagramm

//TODO: UML Flussdiagramm Frontend (client-flow-diagram.png)

## 6.6 Funktionale Anforderungen

Das Frontend ist betroffen von 3 obligatorischen funktionalen Anforderungen. Diese sind:

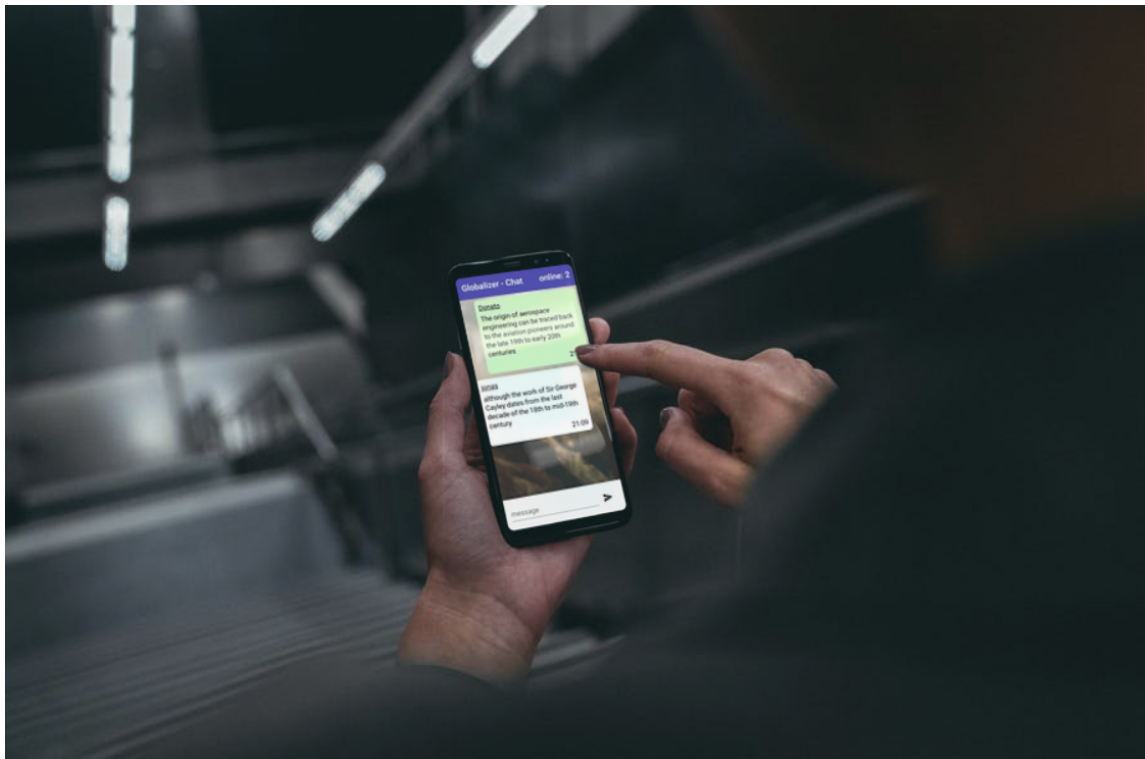
-  USER\_STANDARD\_LOGIN
-  NACHRICHT\_SENDEN
-  NACHRICHT\_EMPFANGEN

Was diese Anforderungen konkret bedeuten kann im Dokument "Anforderungsspezifikationen" nachgelesen werden. Dort sind diese inklusiv Use-Case-Diagramm dargestellt.

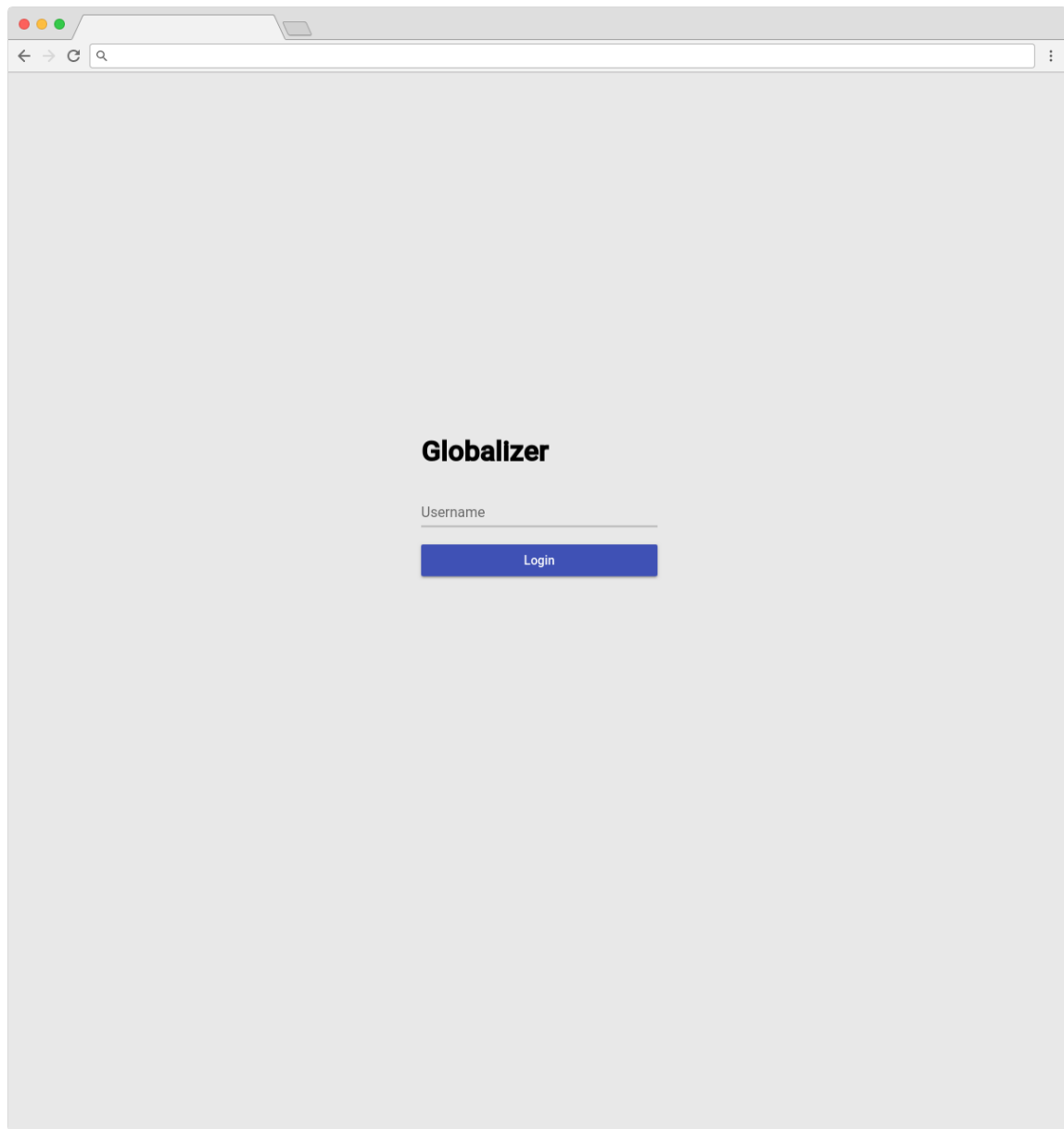
## 6.7 Schnittstellen

Die Schnittstelle zwischen Benutzer und Frontend ist rein visuell. Diese soll sp  ter so aussehen:

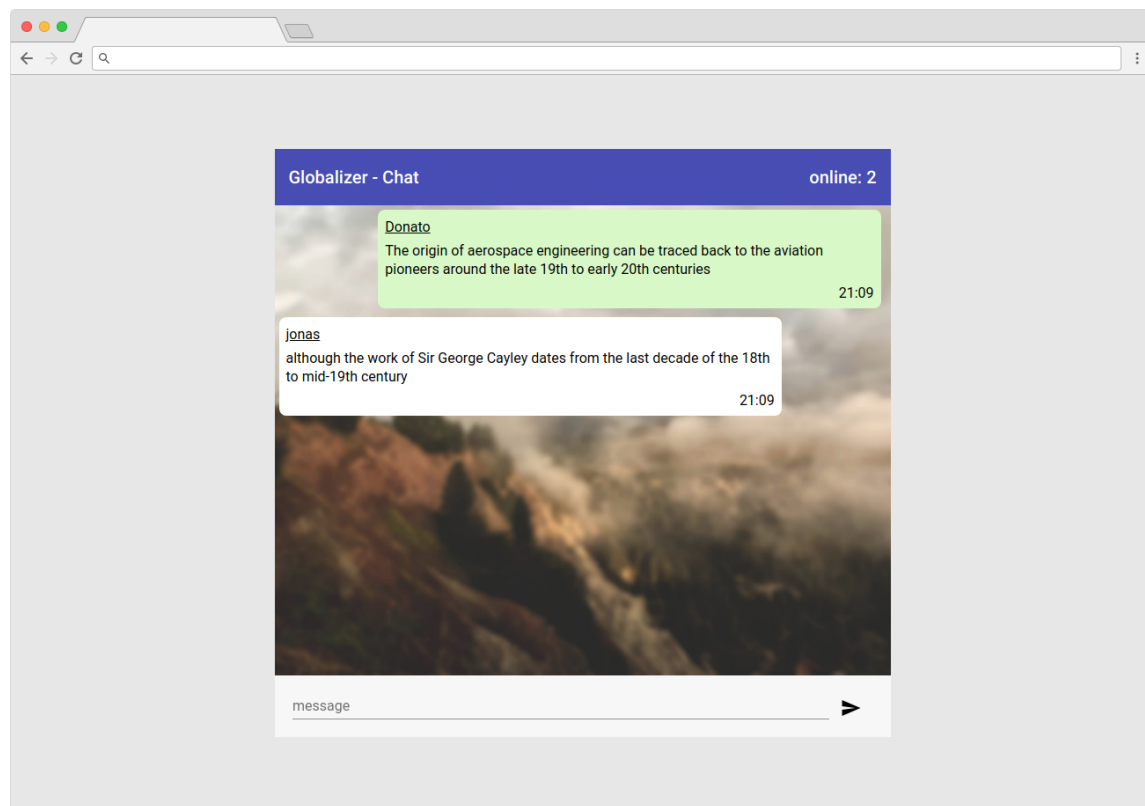
### 6.7.1 Android frame



### 6.7.2 Desktop Chrome Login



### 6.7.3 Desktop Chrome Chat



## 6.8 Persistenz

Das Frontend hat nur einen ganz minimalen Teil an Persistenz. Dazu wird der durch HTML 5 neu eingeführte LocalStorage eingesetzt. Es wird nach dem Login eines Benutzers dessen ID gesichert, damit später ein Sessionlogin durchgeführt werden kann und der Nutzer sich nicht erneut anmelden muss. Da diese Daten aber auf dem Client selbst gespeichert sind, müssen sie mit vorsicht behandelt werden. Der LocalStorage ist ein simpler Key-Value-Store.

## 7 Technologieentscheide

In diesem Abschnitt des Dokuments begründen wir, weshalb wir die von uns gewählten Technologien verwenden.



## 7.1 Webclient

Wir setzen bei der Schnittstelle zwischen Benutzer und System bewusst auf einen Webclient und nicht auf einen Fat-Client. Dies ermöglicht es uns, dass unsere Applikation erreichbar ist, ohne dass zuerst ein Client heruntergeladen und installiert werden muss. Des weiteren können wir so auch direkt Benutzer mit Smartphones abholen, da wir ein Responsive Design verwenden. Es ergeben sich also einige Vorteile durch die Verwendung eines Webclients anstelle des Fat-Clients. Deshalb haben wir uns für diese Variante entschieden.

## 7.2 Angular

Unseren Webclient setzen wir mit Angular 7 um. Dies hat zum einen den Grund, dass wir beide schon Erfahrungen mit Angular 7 haben, zum anderen aber auch, da Angular momentan eine der meist genutzten und am besten geeigneten Frontend-Frameworks ist. Für die Version 7 haben wir uns entschieden, da diese eine viel bessere Kompression als die Vorgängerversionen hat. Somit wird die Datenübertragung minimiert und die Geschwindigkeit der Applikation steigt.

## 7.3 NodeJS

Wir haben uns für NodeJS entschieden, da es sich hier um eine neue und sehr schnelle Sprache handelt. Wir wollen einen möglichst wartbaren Code. Hier ist Node optimal, da es nur einen minimalen Overhead mit sich bringt. So kann unser Server flach gebaut werden ohne grosse Strukturen um diesen herum zu benötigen. Weitere Punkte die für Node sprechen sind, dass unser Front- und Backend somit die gleiche Sprache verwenden (JavaScript) und die Unterstützung für Websockets hier sehr gross ist, dank einer Vielzahl von verfügbaren Libraries.

## 7.4 Github

Als Code-Versionskontrolle haben wir uns für Git entschieden, da dies in der heutigen Zeit der Standard ist und wir beide Git bereits kennen und benutzt haben. Als Ablage-Plattform haben wir uns für Github entschieden, da wir auch beide hier bereits Erfahrungen hatten und Github in unserem Fall das kostengünstigste Angebot ist. Da das Projekt die Anforderung hat, dass es Open Source sein soll, ist Github eine passende Plattform, da auch viele andere grosse Open Source Projekte auf Github setzen.

## 7.5 LaTeX

Der letzte erwähnenswerte Technologieentscheid ist der des Dokumentationstools. Wir verwenden LaTeX, welches geeignet ist um technische und wissenschaftliche Dokumente zu erstellen. Bis jetzt hat noch niemand von uns mit LaTeX gearbeitet. Wir haben uns aber für dieses entschieden nach einer intensiven Internetrecherche und einem ersten Testlauf, welchen wir durchgeführt haben.

# 8 Continuous Integration und Continuous Testing

Wir werden bei unserem Projekt eine Continuous Integration und ein Continuous Deployment einrichten. Es wird zudem auch ein Continuous Testing geben.

## 8.1 CI mit CircleCI

CircleCI ist ein Buildserver, welcher für Continuous Integration genutzt wird. Man kann CircleCI mit Github verbinden und bei jedem Push einen Build "triggern". Somit sieht man direkt, wenn nach einem Push das Projekt nicht mehr gebaut werden kann und Fehler werden sehr frühzeitig erkannt. Dies ermöglicht es uns, dass der Stand welcher auf Github ist immer lauffähig und funktionell ist. CircleCI kann für Open Source Projekte kostenlos genutzt werden. Zu finden unter <https://circleci.com/>.

## 8.2 CT mit Cypress

Cypress ist ein weiteres Tool, welches mit Github verbunden werden kann. Es hat zudem auch eine Anbindung an CircleCI. Wir werden Cypress so konfigurieren, dass bei einem Push ein CircleCI-Build "getriggered" wird und Cypress sich danach in diesen Build einhängt. Cypress soll danach automatisch alle E2E (End-to-End) Tests laufen lassen. Von den E2E Tests wird ein Video aufgezeichnet, welches dann auf Cypress einsehbar ist. So können wir anhand von den Videos auch Fehler finden, welche in den E2E Tests vorkommen. Cypress ist also ein Tool, welches uns ein weiteres Werkzeug zur Überprüfung der Integrität unserer Software geben wird. Cypress ist zu finden unter <https://www.cypress.io/>.

## 8.3 Code Review mit CodeFactor

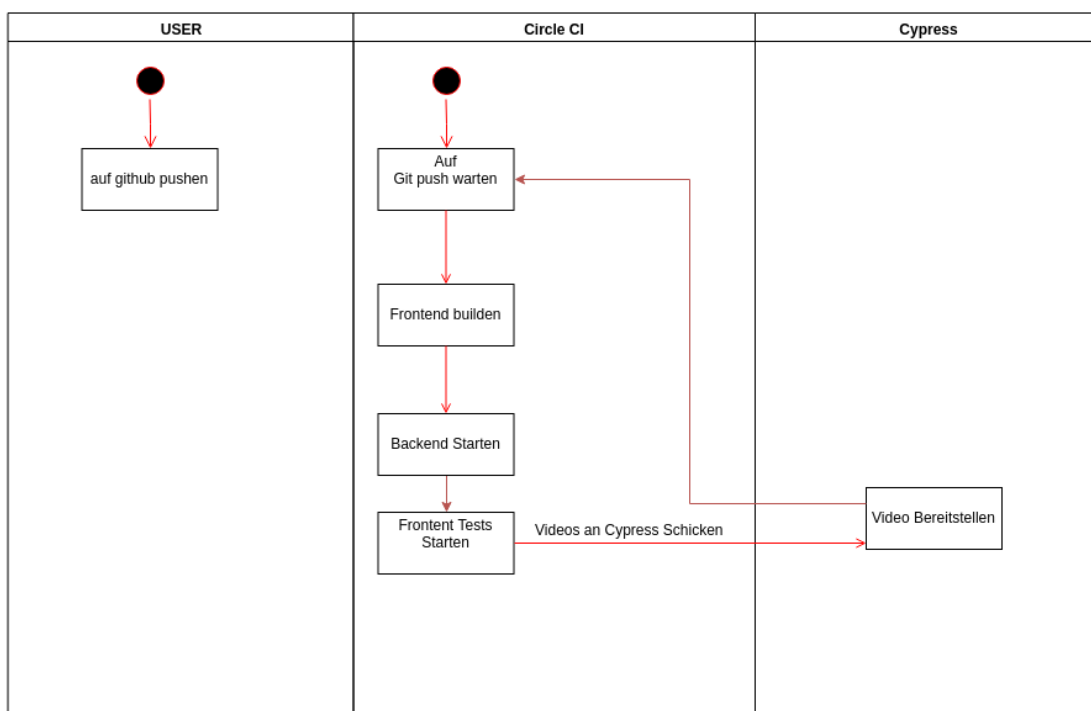
CodeFactor ist ein Tool, welches ebenfalls mit Github verbunden wird. Bei einem Push überprüft es automatisch, ob im Code Sicherheitslücken, Codestyle-Fehler oder

sonstige Fehlerquellen sind. Das ganze kann dann übersichtlich auf einer Weboberfläche von CodeFactor angeschaut werden. Dieses Tool wird uns helfen den Code sauber zu behalten, damit dieser gut wartbar wird. CodeFactor ist zu finden unter <https://www.codefactor.io/>.

## 8.4 Darstellung CI & CT

Der Ablauf von unserem CI & CT sieht wie folgt aus.

### Testing Pipeline



Technologieentscheide //TODO: Hosting bei Heroku  
Deployment //TODO: Deploymentdiagramm bild einfügen