

**«Familie Thunberg»**

**«Suffizienzplattform»**

**Informationen für die Jury**

Inhalt

[1 Zugänge 3](#_Toc17541529)

[1.1 Aktueller Stand des Sourcecodes 3](#_Toc17541530)

[1.2 CI/CD Pipeline 3](#_Toc17541531)

[1.3 Production 3](#_Toc17541532)

[2 Fachliche Aspekte 4](#_Toc17541533)

[2.1 Ausgangslage 4](#_Toc17541534)

[2.2 Umgesetzte Use Cases 4](#_Toc17541535)

[2.3 Technisch Angedachte Use Cases 4](#_Toc17541536)

[2.4 Abgrenzungen 4](#_Toc17541537)

[3 Technische Aspekte 5](#_Toc17541538)

[3.1 Architekturentscheide 5](#_Toc17541539)

[3.2 Bausteinsicht 6](#_Toc17541540)

[3.3 Verteilungssicht 6](#_Toc17541541)

[3.4 Implementation 7](#_Toc17541542)

[4 Verzeichnisse 8](#_Toc17541543)

# Zugänge

## Aktueller Stand des Sourcecodes

* <https://github.com/baernhaeckt> [Organisation]
* <https://github.com/baernhaeckt/Backend> [Backend Source]
* <https://github.com/baernhaeckt/Frontend> [Frontend Source]
* <https://github.com/baernhaeckt/Misc> [Automated Tests, Helper Scripts, Doku]

## CI/CD Pipeline

* <https://dev.azure.com/baernhaeckt/BaernHaeckt2019/_dashboards/dashboard/2a5704d4-5069-4833-9704-ef6f9cde3cd2> [Dashboard]
* <https://dev.azure.com/baernhaeckt/BaernHaeckt2019/_build?definitionId=1> [Backend Build]
* <https://dev.azure.com/baernhaeckt/BaernHaeckt2019/_build?definitionId=2> [Frontend Build]
* <https://dev.azure.com/baernhaeckt/BaernHaeckt2019/_release?_a=releases&view=mine&definitionId=1> [Backend Release]
* <https://dev.azure.com/baernhaeckt/BaernHaeckt2019/_release?_a=releases&view=mine&definitionId=2> [Frontend Release]

## Production

* <http://baernhaeckt.azurewebsites.net/swagger> [Backend – Swagger UI]
* <http://baernhaeckt.azurewebsites.net/swagger/v1/swagger.json> [Backend – Swagger Defintion]
* [https://baernhaeckt.z16.web.core.windows.net](https://baernhaeckt.z16.web.core.windows.net/) [Frontend]

# Fachliche Aspekte

Dieses Kapitel gibt einen Überblick, über die fachlichen Aspekte der Lösung und die Überlegungen, welche angestellt wurden.

## Ausgangslage

Im Rahmen dieser Challenge soll eine Sozialeplattform [[1]](#footnote-1)erstellt werden. Eine Sozialeplattform lebt von ihren Nutzern. Wenn keine kritische Masse erreicht wird, dann wird die Plattform den Durchbruch nicht schaffen. Aus diesem Grund muss das *Onboarding* möglichst einfach sein. Damit die Plattform lebt, müssen die Benutzer mit ihr interagieren. Die Interaktion bzw. die Datenerfassung müssen deshalb *einfach und schnell möglich* sein. Die Sozialforschung sagt uns, dass Menschen sich an ihrem Umfeld messen und dann danach streben, in diesem Umfeld zu den besten zu gehören. Ein *Vergleich mit Freunden und den Umliegenden Benutzern* bringt diesen Ansporn. Mittels *Gamification* werden die Benutzer bei Laune gehalten und für ihre Bemühungen belohnt. Diese Bemühungen sind natürlich auf für die Freunde sichtbar!

## Umgesetzte Use Cases

TODO

## Technisch Angedachte Use Cases

TODO

## Abgrenzungen

* Welche Abgrenzungen habt ihr bewusst vorgenommen und damit nicht implementiert? Weshalb?

# Technische Aspekte

Das folgende Kapitel zeigt den Aufbau des «Ökogram» Software-Systems. Zuerst werden Architekturentscheide erläutert, welche signifikanten Einfluss auf die Lösung haben. Danach wird die Baustein- und Verteilungssicht genauer erklärt.

## Architekturentscheide

1. Die Applikation wird als Webapplikation realisiert. Das ermöglicht eine saubere Trennung zwischen Front- und Backend und dadurch kann die Anzahl Entwickler gut skaliert werden. Des Weiteren ist eine Plattform Unabhängigkeit inhärent gegeben und muss nicht erzeugt werden (wie das z.B. bei Apps der Fall ist). Die Lösung kann zudem schnell und automatisch über eine CI/CD-Pipeline in die Produktion deployt werden.
2. Aus den oben bereits genannten Gründen erfolgt eine Trennung zwischen Frontend und Backend. Das Frontend wird als SPA implementiert, so wird ein gutes Benutzererlebnis erzeugt. Das Backend bietet die Funktionalität ausschliesslich über APIs an. Mit dieser Entkopplung wird die Evolution einfacher (z.B. parallel neues UI v2 und/oder Apps). Die Schnittstellen sind Hauptsächlich als Web APIs (REST & JSON) realisiert. Wo «push» vom Server nötig ist, werden Websockets eingesetzt.
3. Das Frontend wird mit VUE.JS realisiert. VUE.JS ermöglicht eine «progressive» Entwicklung. Das Framework lässt sich am Anfang leichtgewichtig einsetzen und lässt ein rasches Prototyping zu. Wird die Lösung grösser, skaliert das Framework mit und ermöglicht weiterhin die Umsetzung von wartbaren Lösungen. Aktuell wird JavaScript eingesetzt. Wächst die Lösung, ist der Umstieg auf TypeScript möglich.
4. Das Backend wird mit ASP.NET Core mit der Programmiersprache C# realisiert. Bei ASP.NET Core handelt es sich um ein Plattformunabhängiges «cloud native» Web Framework. Das Framework lässt sich mit seinem Middleware-Konzept sehr leicht an diverse Bedürfnisse anpassen. Standardszenarien sind ohne grossen Aufwand abgedeckt. Es ist zudem auf Durchsatz optimiert.
5. Die Persistenz wird mittels einer DocumentDb umgesetzt. Da rasch eine funktionsfähige Lösung entstehen soll, ist der Entscheid zugunsten einer schemalosen DocumentDb gefallen. Aufgrund der vorhandenen Vorkenntnisse wurde der Entscheid zugunsten von MongoDb getroffen.
6. Alle Komponenten werden in der Microsoft Cloud Azure gehostet. Dabei wird überall auf dem höchst möglichen Abstraktionlayer (z.B. SaaS) gearbeitet, um initialaufwand zu vermeiden.
7. Folgende Entscheide sind aufgrund der vorherrschenden Verbreitung der jeweiligen Technologien und dem Vorwissen des Teams gefallen:
   1. API Dokumentation: Swagger & Swagger UI
   2. Authentication: Bearer Token & JWT
   3. Automatisierte Tests & Helper: PowerShell Scripts
   4. Styling: Bootstrap
   5. Frontend http call: Axios

## Bausteinsicht

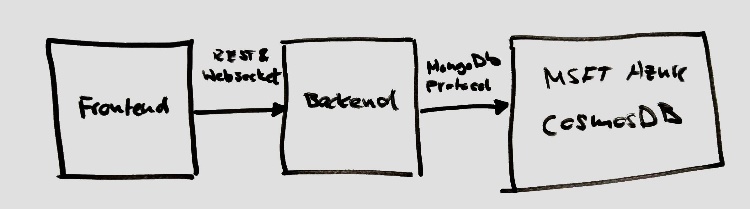


Abbildung 1 Bausteinsicht

Frontend: Das Frontend wurde mittels VUE.JS umgesetzt. Der Aufbau folgt dem Vue.js standard. Es werden folgende Konzepte eingesetzt: Single-file-Components, vue-js-router, vuex. Für das Styling kommt bootstrap und sass zum Einsatz.  
  
Backend: Das Frontend wird als eine einzelne Applikation umgesetzt. Es wird ASP.NET MVC eingesetzt. Die Lösung wird in Schichten gegliedert (wobei keine strikte Trennung erfolgt, es dürfen lediglich keine Referenzen von tieferen Schichten auf höhere Schichten erfolgen). Die «Web» Schicht stellt mittels Controller die REST-Endpoints bereit. Die Logik befindet sich im «Core» und wird in Services gekapselt. Der Datenbank Zugriff wird mit einer dünner Abstraktionsschicht (eine Library wird verwendet) gekapselt. Die Entitäten befinden sich im Projekt «Database».  
  
Persistenz: Die CosmosDb von Microsoft Azure unterstützt die MongoDb API und das MongoDb Protokoll. Sie kann als SaaS bezogen werden und wurde deshalb ausgewählt. Sie enthält eine Datenbank und folgende Collections: Users, TODO

## Verteilungssicht

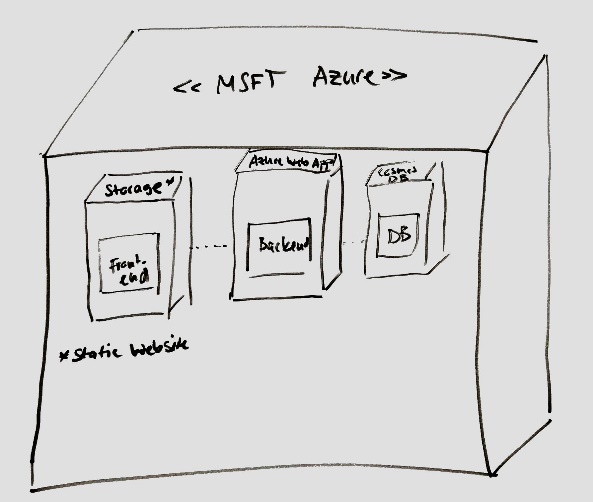


Abbildung 2 Verteilungssicht

Die gesamte Lösung wird auf Microsoft Azure gehostet. Der Entscheid ist aufgrund von Vorwissen gefallen. Grundsätzlich wurde darauf geachtet, dass kein Vendor-Lock-In entsteht. Alle Komponenten lassen sich auch On-Premise oder in einer anderen Cloud betreiben.

Frontend: Azure Storage stellt die Möglichkeit bereit, sehr einfach und günstig eine statische Website zu hosten. Es muss Infrastruktur betrieben werden.

Backend: Azure Web App stellt eine Hosting Umgebung für verschiedene Technologien zur Verfügung. Es musst keine Infrastruktur betrieben werden.

MongoDb: Bei CosmosDb handelt es sich um eine vielseitige DocumentDb, welche verschiedene Protokolle und APIs unterstützt. Es handelt sich um eine SaaS-Lösung und es muss keine Infrastruktur betrieben werden.

## Implementation

* Gibt es etwas Spezielles, was ihr zur Implementation erwähnen wollt?
* Was ist aus technischer Sicht besonders cool an eurer Lösung?

# Verzeichnisse

[Abbildung 1 Bausteinsicht 6](#_Toc17541878)

[Abbildung 2 Verteilungssicht 6](#_Toc17541879)

1. <https://www.bernhackt.ch/challenges-2019-2/#bfh> [24.08.19 [↑](#footnote-ref-1)