dies, dass beim Aufruf der Plattform eine Anfrage an den Controller geschickt wird und dieser anschließend die kompilierte React Anwendung ausgibt. Diese lädt eigenständig jegliche Informationen über HTTP Anfragen und kümmert sich um interne Seitenaufrufe. Die Anfragen werden dabei asynchron an die Rails Applikation geschickt und vom Controller beantwortet. Somit erhält die React Anwendung über die Kommunikation von Controller und Modul, den Content aus der Datenbank, der vorher eingepflegt wurde (Siehe Abbildung).

Demnach gibt es eine Aufteilung hinsichtlich der graphischen Darstellung in Content Management System und Plattform. Die Datenspeicherung und Verwaltung ist jedoch gleich. Auch wird die gesamte Codebase in einem Bereitstellungsprozesses dem Hosting Service bereit gestellt. Somit handelt es sich beim System von PluraPolit um ein Monolith, welches teilweise Modular ist, die REST-Standards erfüllt und nach dem Model-View-Controller-Ansatz aufgeteilt ist. Des Weiteren nutz es vereinzelt externe Services von AWS, um Tondateien zu speichern und Transcripierungen vorzunehmen.

2.2 Microservices

Nachdem nun die einzelne Architekturstile vorgestellt wurden und das System von PluraPolit eingeordnet wurde, wird nun Microservices vorgestellt.

Dabei handelt es sich um eine Architekturstil, der ein Vertreter der verteilten Systeme ist und sich historisch aus der Service Orientierten Architektur abgeleitet hat (siehe Abschnitt 1.2).

Eberhard Wolff beschreibt Microserivces als Ansatz Software in einzelne Module zu teilen und definiert es als Modularisierungskonzept, welches Einfluss auf die Unternehmesorganisation und Software-Entwicklungsprozess hat (vgl. Wolff 2018, Kap. 1.1). Dabei ist jedes Module ein eigenes Programm.

Sam Newman schließt sich Wolff an und beschreibt Microservices als voneinander unabhängig einsetzbare Dienste, die um eine Geschäftsdomäne herum modelliert sind (vgl. Newman 2019, Kap. 2.1).

Somit beschreiben beide Microservices als ein System aus einzelnen unabhängigen Services, die sich an ein Geschäftsdomäne richten. Insbesondere die Abhängigkeit zum Geschäftsprozess wird nachfolgend näher beschrieben.

2.2.1 Zusammenhang und Verknüpfung

Wenn es um das aufteilen von Software geht, ist es wichtig zu verstehen wie die einzelnen Funktionen und Klassen zusammenhängen und welche Verknüpfungen es zwischen ihnen

gibt.

Dabei bezieht sich der Zusammenhang auf die funktionale Abhängigkeit zweier Funktionen oder Klassen (vgl. Newman 2019, Kap. 2.3.1). Somit liegt ein hoher Zusammenhang vor, wenn Quellcode anhand seiner logischen Zugehörigkeit geordnet ist. Eine konkrete Umsetzung dieses Bestreben ist im Abschnitt zum Model-View-Controller-Ansatz zu finden.

Verknüpfung hingegen beschreibt in welchem Maß, Funktionen und Klassen verbunden sind, ohne dass sie logisch zusammen gehören (vgl. Newman 2019, Kap. 2.3.2). Somit bezieht sich Verbindung ausschließlich auf eine technisch vorliegende Kopplung. Ein typisches Beispiel hier für ist, wenn ein Datenabruf an mehreren Stellen direkt über das Datenbankschema abläuft. Dadurch entsteht eine Abhängigkeit auf das Schema, sodass falls sich das Schema ändert auch die jeweiligen Aufrufe geändert werden müssen.

Insbesondere in monolithischen Systemen können viele Verknüpfungen entstehen, da keine festen Abgrenzungen zwischen einzelnen logischen Bereiche definiert sind. Somit kann es sein, dass ein Monolith, welches historisch wächst, keine klare Struktur aufzeigt. Es entsteht hieraus ein System, welches anfällig für Veränderungen ist.

Für ein Microservicesarchtektur ist jedoch das Bestreben klare Abgrenzung zu erlangen, und einzelne stabile Services zu etablieren. Diese sollen soweit es geht von einander entkoppelt funktionieren. Somit ist das Ziel für eine Microservicearchitektur einen hohen Zusammenhang bei geringer Verknüpfung zu besitzen. Dabei ist dies nicht nur die Zielsetzung für Microservices allgemein, sondern ein generelles Bestreben für stabile Systeme.[13] Für Microservices bedeutet dies Konkret, dass ein Services ausschließlich aus funktional abhängigem Quellcode besteht und technische Implementierungen, wie zum Beispiel Datenbankstrukturen und Funktionsaufrufe, hinter klar definierten Schnittstellen versteckt sind.

2.2.2 Das Gesetz von Conway

Wenn es um die Teamaufteilung in IT Projekten geht, wird in den meisten Fällen sich an die historischen Herangehensweise orientiert und die Entwickler hinsichtlich ihrer Spezifikation eingeteilt. So werden Datenbankexperten, Frontend- und Backendentwicklern in jeweils einzelne Teams aufgegliedert.

Entscheidet sich jedoch ein Unternehmen für eine Microservicearchitektur, bedeutet diese Aufteilung, dass für jeden einzelnen Service Absprachen zwischen den Teams entstehen. Folglich etabliert sich eine Menge an einzelnen Kommunikationsbeziehungen, welche die

[13] Dies bezieht sich auf das Gesetz von Constantine, welches besagt *„A structure is stable if cohesion is strong and coupling is low.“* (Endres und Rombach 2003, S. 43).

Umsetzung von neuen Funktionen verlangsamt.

Der amerikanische Mathematiker Melvin Edward Conway umschrieb das Dilemma wie folgt: *„Organisationen, die Systeme designen, können nur solche entwerfen, welche die eigene Kommunikationsstruktur widerspiegelt.“* Er bezieht sich dabei auf die Tatsache, dass die Softwarearchitektur immer von den einzelnen Entwicklern mit entworfen wird und daher eingefahrene Kommunikationsstrukturen [14] sich in der Architektur wiederfinden lassen.

Nach Conway ist somit die Architektur direkt mit der Kommunikationsstruktur verbunden. Folglich sollte ein Unternehmen, welches das Gesamtsystem in einzelne Module teilt auch die gleiche Aufteilung in der Kommunikationsstruktur einführen. Konkret bedeutet dies für ein Unternehmen, dass die Entwickler nicht nach Spezifikation, sondern nach Kontext des Service geteilt werden. Diese Teams besitzen dabei möglichst kurze Kommunikationswege und haben wenig Abhängigkeiten zu anderen Teams. Im genaueren Definiert sich die Verknüpfung zu anderen Teams über die Schnittstellen der Services. Um die Kommunikation für ein Service auf das Minimum zu reduzieren, sollte nur ein Team pro Dient verantwortlich sein, wobei ein Team auch mehrere Services betreuen kann.

Um nun folglich eine Microservicesarchitektur umzusetzen, muss zum einem die Anwendung so Komplex sein, dass sie sich in einzelne Services teilen lässt und zum anderen so viele Entwickler vorhanden sein, dass idealerweise jeder Dienst von einem Team betreut wird.

2.2.3 Bounded Contexts

An diesem Punkt sollte eine Definition des Begriffes Bounded Contexts gegeben werden. Dies ist insbesondere wichtig, da Domain Driven Design darauf aufbaut und Bounded Contexts umsetzt.

2.2.4 Domain Driven Design

- Was ist DDD?
- Was sind Bounded Contexts?
- Bespiele dazu.
- Schlussfolgerung für PluraPolit

[14] Kommunikationsstrukturen ist dabei nicht zwingen gleich das Organigramm, da etabliere Kommunikationswege nicht zwingend von der Unternehmensstruktur abgebildet wurde.