THM | CAMPUS GIESSEN | MNI
TECHNISCHE HOCHSCHULE MITTELHESSEN
Mathematik, Naturwissenschaften und Informatik

Bachelorarbeit

Modularisierung einer Single-Page-Application mittels Module Federation

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

vorgelegt dem
Fachbereich Mathematik, Naturwissenschaften und Informatik
der Technischen Hochschule Mittelhessen
von

Timon Pellekoorne

1. Dezember 2021

Referent: Prof. Dr. Frank Kammer
Korreferent: Prof. Thomas Friedl

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 1. Dezember 2021

Zusammenfassung

Für Anwendungen, welche einen großen Funktionsumfang besitzen, bringt die Modularisierung einige Vorteile mit sich. So kann ein Nutzer die Anwendungen individuell an seine Bedürfnisse anpassen. Der Anbieter kann individuelle Preismodelle erstellen und somit einen breiteren Markt mit seinem Produkt ansprechen. Auch kann die Entwicklung an verschiedenen Modulen stattfinden und so den Entwicklungsprozess übersichtlicher gestalten.
Dabei erlaubten es die Technologien für Single-Page-Applications lange nicht den Code anderer Anwendungen zu laden. Dies ist jedoch notwendig, um ein modulares System zu entwickeln, in welchem Module unabhängig entwickelt werden sollen. Aushilfe schafft dabei die am 10. Oktober 2020 veröffentlichte Technologie *Module Federation*. Diese bietet die Möglichkeit, Code zwischen mehreren Anwendungen zu teilen und eröffnet somit ganz neue Möglichkeiten für die Modularisierung einer Single-Page-Application.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

In dem ersten Kapitel dieser Arbeit wird eine Einleitung in das der Arbeit zugrunde liegende Thema gegeben. Dazu wird zunächst auf die Motivation der Arbeit eingegangen, um daraufhin die Webanwendung *Audimax* vorzustellen, auf deren Grundlage die Konzeptionierung und Implementierung aufbaut. Abschließend wird die Zielsetzung dieser Arbeit sowie die Methodik der Arbeit erläutert.

1.1 Motivation

Durch die Digitalisierung steigt die Nachfrage nach immer neuen Anwendungen auch im Bereich der Schulen und Hochschulen. Besonders während der Covid19-Pandemie entstand aufgrund des Wegfallens der Präsenzveranstaltungen der Bedarf einer Kommunikationsplattform, welche alle benötigten Funktionen zur Unterstützung der digitalen Lehre vereint. Da die Integration solcher Anwendungen zu Beginn der Pandemie schnell gehen musste, griffen viele Schulen und Hochschulen auf bereits am Markt angebotene Softwarelösungen zurück. Hierbei wurden vor allem Anwendungen wie *Zoom* für Videokonferenzen, *Microsoft Teams* für die Zusammenarbeit online und – vor allem im Bereich der Grundschulen – *Padlet* eingesetzt. Der Einsatz der genannten Softwarelösungen führte jedoch schnell zu Problemen. So kommt es bei der Nutzung dieser Softwares gerade im Bereich des Datenschutzes zu Schwierigkeiten, da die Anbieter außerhalb der Europäischen Union sitzen und somit nicht an die Datenschutzgrundverordnung der EU gebunden sind. Daher wurde beispielsweise die Nutzung von Padlet durch eine Stellungnahme des hessischen Beauftragten für Datenschutz und Informationsfreiheit am 1. Februar 2021 den Schulen untersagt [fDuI21]. Zusätzlich zu den Bedenken im Bereich des Datenschutzes fehlte auch eine Software, welche alle benötigten Funktionen vereint. So konnte es vorkommen, dass an einer Hochschule für die Dateiverwaltung *Moodle* genutzt wurde, für Videokonferenzen *Zoom* und für die Kommunikation außerhalb von Lehrveranstaltungen *Discord*.

Aus diesen Gründen startete Ende Oktober die Entwicklung einer neuen Webanwendung mit dem Namen *Audimax*, die das Ziel verfolgt, den Bedarf an Programmen, welche die digitale Lehre unterstützen, in einer einzigen Anwendung zu vereinen.
Während der Entwicklung von Audimax und dem Einsatz an Hochschulen sowie Grundschulen kam der Bedarf nach einer modularen Anpassung der Anwendung auf. Hierbei benötigen nicht alle Mandanten die gleichen Funktionen, weshalb es möglich sein sollte, diese je nach Bedarf zu aktivieren oder deaktivieren. Diese Funktionalität bietet zusätzlich dem Anbieter die Möglichkeit, individuelle Preismodelle zu bilden, sodass jeder Kunde nur für die benötigten Funktionen zahlt. Des Weiteren muss in der Benutzeroberfläche lediglich

der Code von benötigten Funktionen geladen werden. Dies führt zu kürzeren Ladezeiten und somit zu einer benutzerfreundlichen Interaktion.
Neben dem Wunsch der individuellen Anpassbarkeit stieg die Nachfrage nach immer neuen Funktionen, sodass ein modularer Aufbau der Anwendung, bei welchem Funktionen als einzelne Module dargestellt werden, den Entwicklungsprozess übersichtlicher gestaltet.

Um diese Funktionalitäten gewährleisten zu können, musste die Architektur der Anwendung angepasst werden. Dies führte gerade bei der Entwicklung der Benutzeroberfläche aufgrund des Aufbaus der Anwendung zu Besonderheiten, welche den Inhalt dieser Arbeit bilden und im Nachfolgenden genauer erläutert werden.

1.2 Audimax

Aufbau

Die oberste Ebene der Organisation in Audimax stellen *Kurse* dar. Kurse können durch Benutzer erstellt werden. Beim Erstellen eines Kurses – oder auch im Nachhinein – können diesem Benutzer hinzugefügt werden, welche innerhalb eines Kurses verschiedene Rollen besitzen können. Die Hauptfunktionalitäten innerhalb eines Kurses stellen die sogenannten *Kanäle* dar. Von diesen Kanälen können beliebig viele innerhalb eines Kurses angelegt werden, wobei Kanäle von verschiedenen Typen sein können. Ein Kanaltyp repräsentiert hierbei eine Funktion in Audimax. So gibt es zum Beispiel den Kanaltyp *Konferenz*, welcher beim Klicken auf diesen Kanal eine Konferenz bereitstellt. Dargestellt werden die Kanäle durch eine Ordnerstruktur für die bessere Verwaltung aller Funktionen (siehe Abbildung 1.1).

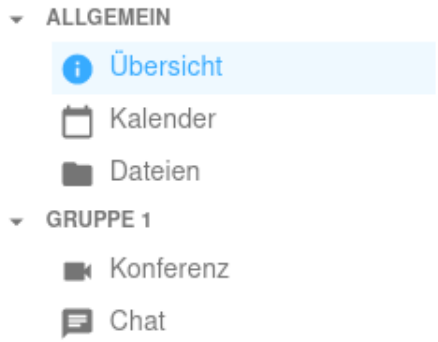

Abbildung 1.1: Ansicht der Kanäle in einem Kurs

Ist-Zustand

Audimax baut im Backend auf mehreren Open-Source-Softwares auf. So nutzt Audimax zum Beispiel die Software *BigBlueButton* für das Anbieten von Videokonferenzen oder

auch *Matrix* für die Implementierung eines Chat-Systems. Diese Systeme werden für den Benutzer in einer React-Single-Page-Application vereint. Dabei kommuniziert diese mit den einzelnen Diensten im Backend.

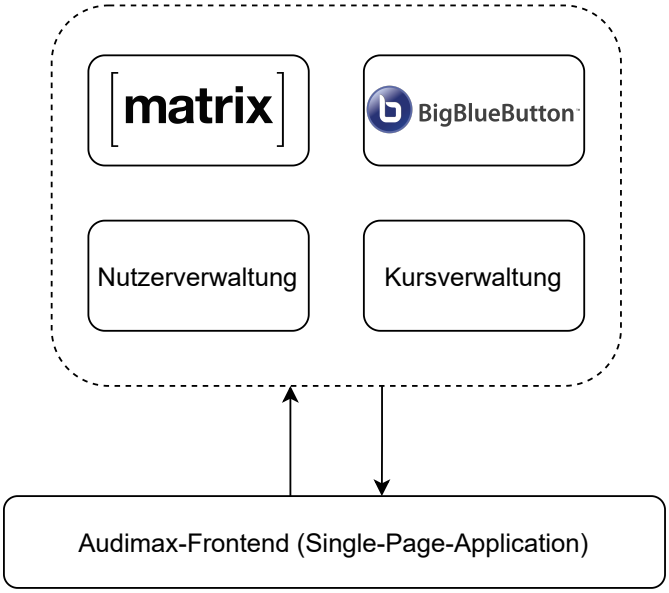

Abbildung 1.2: Aufbau Audimax

Die Single-Page-Application stellt eine einzelne Anwendung dar und ist somit nicht modular. Um diese zu modularisieren, müsste vom Frontend bei Bedarf der nötige Code nachgeladen werden, um diesen auszuführen und die Benutzeroberfläche zu erweitern. Dies stellt jedoch bei Single-Page-Applications eine gewisse Herausforderung dar, da die Technologien von Single-Page-Applications das Laden von Code anderer Anwendungen lange nicht unterstützten.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, ein Konzept zu entwickeln, welches die Modularisierung einer Single-Page-Application ermöglicht. Dabei beschränkt sich die Arbeit nicht nur auf das Aufteilen des Codes in einzelne Abschnitte, welche dann dynamisch geladen werden. Vielmehr soll ein System entwickelt werden, welche die Entwicklung von Erweiterungen, ähnlich denen bekannter Anwendungen wie zum Beispiel WordPress oder Visual Studio Code, ermöglicht. Dabei stellen sich folgende Fragen: Können Konzepte aus vorhandenen Plugin-Systemen für die Konzeptionierung einer modularen Single-Page-Application übernommen werden? Welches der Systeme eignet sich am besten für das Konzept der Modularisierung einer Single-Page-Application? Kommt es dabei zu Einschränkungen?
In Bezug auf die beschriebene Kommunikationsplattform *Audimax* ist das Ziel der Arbeit,

die Funktionalitäten in eigene Module auszulagern, sodass zum Beispiel die zuvor beschriebenen Kanaltypen erst dann verfügbar sind, wenn das dazugehörige Modul aktiviert ist. Dabei soll in Kapitel 4 beispielhaft die Funktion *Konferenz* in ein Modul ausgelagert werden.

Für die Umsetzung des erarbeiteten Konzepts soll das am 10.10.2020 mit *Webpack 5* veröffentlichte Feature *Module Federation* genutzt werden.

1.4 Methodik

In Kapitel 2 werden zunächst die Technologien vorgestellt, welche ein besseres Verständnis der Arbeit bieten. Die in diesem Kapitel beschriebenen Technologien bilden die Grundlage der Konzeptionierung und Implementierung der Modularisierung.

In Kapitel 3 wird das Konzept erarbeitet, welches die Modularisierung von Single-Page-Applications ermöglicht. Hierzu werden generelle Konzepte zu erweiterbarer Software sowie der Aufbau der eigenen modularen Anwendung genauer erläutert. Zuletzt wird in diesem Kapitel die Technologie *Module Federation* beschrieben, welche für die Umsetzung des in diesem Kapitel entwickelten Konzepts essenziell ist.

Nachdem das Konzept der Modularisierung in Kapitel 3 erarbeitet wurde, wird in Kapitel 4 die Implementierung dessen in die Anwendung Audimax erklärt. Hierbei wird mithilfe von Code-Beispielen eine praxisnahe Umsetzung der einzelnen Komponenten des Konzeptes beschrieben.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Technologien erläutert, welche zum besseren Verständnis der Arbeit benötigt werden.

2.1 Single-Page-Application

Zu Beginn des Internets bestanden Webanwendungen lediglich aus mehreren HTML-Dokumenten, welche beim Aufruf einer URL mittels eines HTTP-Request vom Server an den Client übermittelt wurden. Diese HTML-Dokumente konnten daraufhin im Browser angezeigt werden [vgl. Jaz07, S. 1]. Solche Webanwendungen waren statisch. Veränderungen konnten nur vorgenommen werden, indem ein Entwickler die tatsächlichen HTML-Dokumente veränderte.
Über die Zeit wurde jedoch der Bedarf nach dynamischen Webanwendungen immer größer, sodass weiterhin vom Server HTML-Dokumente zurückgegeben wurden, jedoch diese serverseitig dynamisch erzeugt wurden. Änderungen der Benutzeroberfläche konnten so jedoch weiterhin lediglich über das erneute Laden des gesamten HTML-Dokumentes durchgeführt werden.
Um das Nachladen oder Generieren von Inhalten zu ermöglichen und die Benutzerinteraktion zu verbessern, wurde 1995 JavaScript von Netscape veröffentlicht. Dabei war JavaScript nicht die einzige Technologie für eine bessere Benutzerinteraktion. So stellte Java seit der Version 1.0 sogenannte *Java Applets* zur Verfügung, mit welchen es möglich war, kleine in Java geschriebene Programme über den Browser auszuführen. Auch wurde lange der *Flash Player* von Adobe genutzt, welcher es ermöglichte, Flash-Anwendungen im Browser auszuführen. Da über die Zeit JavaScript alle diese Technologien obsolet machte, setzte sich schlussendlich JavaScript als Standard für den Browser durch.
JavaScript ist eine clientseitige Skriptsprache, die von einem Browser ausgeführt werden kann. Dadurch ist eine Ausführung von JavaScript-Code auch ohne Internet möglich, da der Browser einen Interpreter für JavaScript beinhaltet. Durch JavaScript wurde es möglich, zum Beispiel Eingabefelder von Formularen schon im Browser auf Korrektheit zu überprüfen, um so die Werte erst nach der Überprüfung an den Server zu senden. [vgl. Bue18, S. 32] Der Einsatz von JavaScript wurde über die Zeit immer umfangreicher, sodass immer mehr Logik, welche vorher auf dem Server ausgeführt wurde, auf der Seite des Clients implementiert wurde. Dies ging so weit, dass vom Server lediglich ein einziges HTML-Dokument geladen wurde und der gesamte Inhalt mithilfe von JavaScript in das eine Dokument geladen wird. Eine solche Anwendung ist eine Single-Page-Application. Der Vorteil solcher Anwendungen ist es, dass sie ganz unabhängig von einem Server funktionieren

können, sobald sie einmal geladen wurden. Auch ist die Interaktion in den Anwendungen reibungsloser möglich und bietet somit dem Benutzer das Gefühl, sich in einer auf dem Betriebssystem installierten Anwendung zu befinden. Dies wird zusätzlich durch die Möglichkeit eines Browsers unterstützt, geladene HTML-Dokumente sowie JavaScript-Dateien zu *cachen*, sodass Anwendungen, – solange diese keine Daten von einem Server benötigen – auch ohne Internetverbindung funktionieren können. Wird beispielsweise ein Taschenrechner mittels einer Single-Page-Application entwickelt, so kann die ganze Berechnung sowie die Anpassungen der Benutzeroberfläche im Browser ausgeführt werden. Somit wäre dieser nach erstmaligem Laden der benötigten Dateien ohne Internet verfügbar.

2.2 TypeScript

TypeScript ist eine von Microsoft entwickelte Programmiersprache, welche auf JavaScript aufbaut. Hierbei wird JavaScript um weitere Sprachfeatures, wie zum Beispiel einem statischen Type-Checker, Sichtbarkeiten innerhalb von Klassen, aber auch Interfaces, welche von Klassen implementiert werden können, erweitert. [vgl. Har20, S. 363] Da TypeScript nur eine Erweiterung von JavaScript ist, ist jeder gültige JavaScript-Code auch in TypeScript gültig.
Weil ein Browser nur JavaScript ausführen kann, muss der TypeScript-Code vor der Ausführung durch den TypeScript-Compiler zu gültigem JavaScript-Code compiliert werden.
Der wohl größte Unterschied gegenüber der dynamisch getypten Sprache JavaScript ist der Type-Checker. In JavaScript können einer Variablen Werte verschiedener Typen zugewiesen werden. [vgl. Har20, S. 363] Dies kann auf der einen Seite praktisch sein, da sich keine Gedanken um die Typisierung gemacht werden muss, auf der anderen Seite ist dies sehr fehleranfällig. Da keine Überprüfung der Typen in JavaScript stattfindet, könnte einer Variablen, der zunächst eine Zahl zugewiesen wurde, im Laufe des Programmes ein Objekt zugewiesen werden. Soll dann mit dieser Variable eine Berechnung durchgeführt werden, wird dieser Fehler erst zur Laufzeit auffallen.
TypeScript bietet nun die Möglichkeit bei der Deklaration von Variablen, einen oder mehrere Typen anzugeben. Hierbei kann der Typ ein von TypeScript vordefinierter Typ wie zum Beispiel *number* oder *string* sein. Zusätzlich können auch eigene Typen definiert werden. Es müssen jedoch nicht zwangsläufig Typen in TypeScript angegeben werden. In diesem Fall versucht TypeScript eigenständig herauszufinden, von welchem Typ eine Variable ist, indem zum Beispiel der Typ des Wertes der ersten Zuweisung geprüft wird. Dieses Verfahren wird *Type Inference* genannt. [vgl. Har20, S. 366]
Um Variablen eigene Typen zu vergeben, gibt es in TypeScript mehrere Möglichkeiten. So kann mithilfe des Schlüsselwortes *type* ein neuer Typ definiert werden, oder es wird ein Interface definiert, welches ebenfalls als Typ angegeben werden kann. Dabei ist die Wahl zwischen der Verwendung eines Typs oder eines Interfaces häufig unerheblich. So entspricht der in Listing 1 definierte Typ dem ebenfalls dort gezeigten Interface.

In beiden Fällen kann eine Klasse diesen Typen implementieren oder einer Variablen kann dieser, wie in Listing 2 gezeigt, als Typ gegeben werden. Dabei würde jedoch in diesem

```
interface Person {
  name: string;
  age: number;
  email?: string;
}
```

```
type Person = {
  name: string;
  age: number;
  email?: string;
}
```

Listing 1: Beispiel: Type und Interface Definition

Beispiel der TypeScript-Compiler einen Fehler ausgeben, da bei der Zuweisung der Variablen lediglich der Name angegeben wurde, hier aber durch die Definition ebenfalls das Feld *age* angegeben werden muss. Das Feld *email* hingegen kann leer bleiben, da durch das im Typ angegebene Fragezeichen dieses als optional markiert wurde.

```
let person: Person = {name: "Max"};
```

Listing 2: Beispiel: Definition einer getypten Variablen

2.3 React

React ist neben AngularJS und VueJS eines der drei großen JavaScript Webframeworks. In einer von *Stackoverflow* geführten Nutzerumfrage, ist React mit 40,14 % das beliebteste Webframework [Ove21]. Entwickelt wird React von Facebook und wird unter anderem von Netflix, Amazon und Airbnb verwendet. [vgl. Har20, S. 3]

Mithilfe von React lassen sich Single-Page-Applications entwickeln. Die Grundstruktur dafür bilden Komponenten, welche gemeinsam als Komposition in ein einziges HTML-Dokument gerendert werden. [vgl. Har20, S. 4] Eine einzelne React-Komponente vereint dabei die Logik und die Darstellung. Dies ist durch die React-eigene Spracherweiterung *jsx* möglich. Hierbei bietet jsx die Möglichkeit, HTML-Code innerhalb von JavaScript-Code zu schreiben. React-Komponenten können dabei entweder als Klassen oder als Funktionen geschrieben werden. Bei der Implementierung einer Komponente als Klasse muss diese die von React zur Verfügung gestellte Klasse *Component* erweitern. Wird hingegen eine Komponente als Funktion implementiert, so ist diese vom Typen *FunctionComponent*. Da bei der Implementierung einer Klassenkomponente die Oberklasse *Component* Funktionen bereitstellt, mit welchen auf React-Features wie zum Beispiel der Zustandsverwaltung oder der Verwaltung des Lebenszyklus einer Komponente zugegriffen werden kann, braucht es eine Möglichkeit auch in Funktionskomponenten auf diese Features zurückzugreifen. Aushilfe bietet hierbei die im Februar 2019 veröffentlichte *Hooks-API* [Abr19]. Da die in dieser Arbeit gezeigten Implementierungen Funktionskomponenten und somit auch die Hooks-API nutzen, wird diese im späteren Teil dieses Abschnittes genauer erläutert.

Komponenten

Eine Funktionskomponente wird als JavaScript-Funktion geschrieben, welche als Übergabeparameter sogenannte *Properties* übergeben bekommen kann, die innerhalb der Komponenten verwendet werden können. Diese Funktion gibt den zu rendernden jsx-Code zurück.

```
function MyComponent(props) {
  return (
    <h1>{props.msg}</h1>
  )
}
```

Listing 3: Funktionale React-Komponente

Die Komponente aus Listing 3 zeigt eine als Parameter übergebene Nachricht an. Aufgerufen wird diese Komponente wie ein eigener HTML-Tag.

```
<MyComponent msg="hallo" />
```

Listing 4: Aufruf einer Komponente

Neben den selbst definierten Properties steht in einer Komponente die Property *children* zur Verfügung. Umschließt eine Komponente weitere Komponenten, kann über die Property children darauf zugegriffen werden, um diese zu rendern.

```
<MyComponent msg="hallo" >
  <OtherComponent />
</MyComponent>
```

```
function MyComponent(props) {
  return (
    <h1>{props.msg}</h1>
    {props.children}
  )
}
```

Listing 5: Rendern der *children* einer Komponente

In dem in Listing 5 gezeigten Beispiel wird über dem Inhalt der Komponente *OtherComponent* die Nachricht „hallo“ gerendert.

Die Spracherweiterung *jsx* bietet viele Vorteile. Einer der Vorteile ist, dass durch die Kombination von HTML- und JavaScript-Code HTML-Elemente oder auch eigene Komponenten aus Objekten gerendert werden können. So kann einer Komponente ein Array aus den Zahlen 1 und 2 angelegt werden, wodurch es möglich ist, dass diese Komponente eine aus diesem Array dynamisch gerenderte Liste mit dem Inhalt des Arrays zurückgibt (Listing 6). Diese Vorgehensweise erlaubt es, die Oberfläche dynamisch aus Objekten rendern zu lassen.

```
const list = [1, 2];

return (
  <ul>
    {list.map((element) => (
      <li>{element}</li>
    ))}
  </ul>
);
```

Output:

- 1
- 2

Listing 6: Generieren von HTML-Code aus einem Array

In React besitzt jede Komponente einen Lebenszyklus. Dieser Zyklus umfasst den Zeitraum vom Hinzufügen der Komponente in den HTML-Code bis zum Entfernen dieser. Dabei kann zu jedem dieser Zeitpunkte in den Prozess eingegriffen werden. Bei Klassenkomponenten stehen dafür unter anderem die Funktionen *componentDidMount*, *componentDidUpdate* und *componentWillUnmount* zur Verfügung.

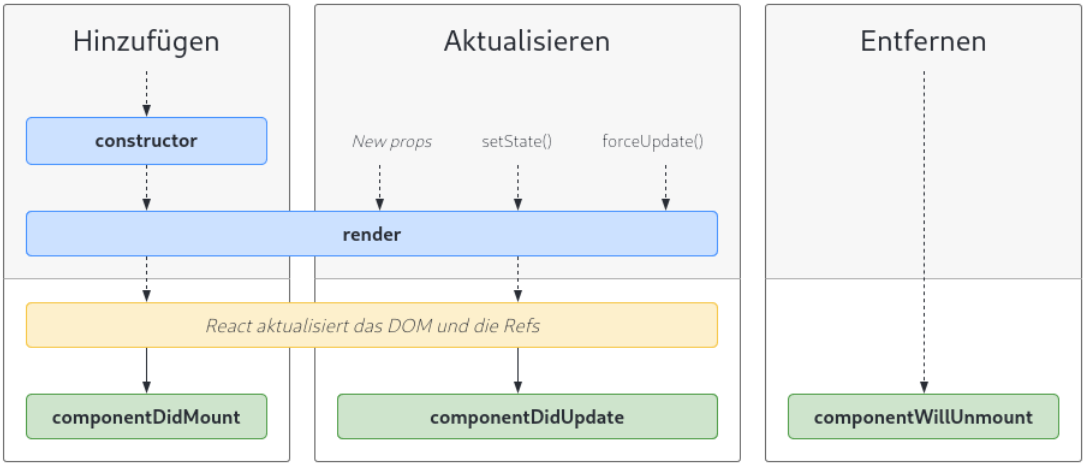

Abbildung 2.1: Lebenszyklus React-Komponente[Maj]

Bei der Verwendung von Funktionskomponenten kann mittels der *useEffect*-Hook auf die Zeitpunkte zugegriffen werden. Die Erläuterung dieser folgt im nächsten Abschnitt.
Ein weiteres Feature von React ist das Halten von Zuständen innerhalb einer Komponente. Ein Zustand (*state*) ist lediglich ein JavaScript Objekt, welches die Besonderheit besitzt, dass bei Änderungen dieses Objektes, also beim Verändern eines Zustandes einer Komponente, alle Komponenten, die dieses Objekt verwenden, neu gerendert werden.

Hooks-API

Die Hooks-Api erweitert Funktionskomponenten um die in Klassenkomponenten durch Funktionen der Oberklasse *Component* gegebenen Features. Zusätzlich können Hooks in React auch selbst geschrieben werden, um mehrfach verwendeten Code auszulagern. Im Folgenden werden zwei Hooks aus der von React zur Verfügung gestellten Hooks-API genauer erläutert.

useState Innerhalb einer Klasse einen Zustand zu verwalten ist kein Problem. So kann in einer Klassenkomponente der Zustand über *this.state* aufgerufen werden und mit der Funktion *this.setState* verändert werden. Beides wird von der Oberklasse bereitgestellt.
Eine Funktion hingegen ist zunächst zustandslos [vgl. Har20, S. 55]. Um einen Zustand zu verwalten, kann mittels der useState Hook solch einer erzeugt werden. Dabei gibt die Hook useState ein Array aus einer Variablen, welche den aktuellen Zustand beinhaltet, sowie eine Funktion, um diesen zu aktualisieren, zurück. Der Vorteil davon, dass ein Array zurückgegeben wird, liegt vor allem darin, dass der Zustand sowie die Funktion frei benannt werden können. Der Hook kann auch ein initialer Wert übergeben werden, welcher beim ersten Rendern der Komponente gesetzt wird.

```
//Langschreibweise
const stateArray = useState(initialValue);
const state = stateArray[0];
const setState = stateArray[1];

//Kurzschreibweise
const [state, setState] = useState(initialValue);
```

Listing 7: Erzeugung eines Zustands in einer Funktionskomponente

Wird nun der Wert mit der Funktion *setState* (Listing 7) aktualisiert, werden alle Komponenten, welche den Zustand verwenden, erneut gerendert.

useEffect Die useEffect-Hook ersetzt in der Hooks-API die Funktionalitäten der Funktionen in Klassenkomponenten, die für den Zugriff auf verschiedene Zeitpunkte des Lebenszyklus einer Komponente bereitgestellt werden. Dieser Hook wird eine Funktion übergeben, welche bei jeder Aktualisierung der Komponente ausgeführt wird. Somit ersetzt sie die Funktionalität der Funktion *componentDidUpdate*. Da es nicht immer notwendig ist, bei jeder Aktualisierung die Funktion erneut auszuführen – zusätzlich kann dies auch zu Laufzeit-Problemen führen – kann der Hook als zweiter Parameter ein sogenanntes *dependecy array* übergeben werden. In diesem können die Variablen angegeben werden, bei deren Veränderung die Hook ausgeführt werden soll. Wird der Hook ein leeres Array übergeben, so wird die Funktion lediglich beim Hinzufügen der Komponente in den HTML-Code ausgeführt. Dies stellt somit die Funktionsweise der Funktion *componentDidMount* dar. Um

zuletzt noch die Funktionsweise der Funktion *componentWillUnmount* zu ersetzen, kann die der Hook übergebene Funktion eine weitere Funktion zurückgeben. Diese wird beim Entfernen der Komponente ausgeführt.

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

Listing 8: Nutzung der useEffect-Hook [Read]

In dem in Listing 8 gezeigten Beispiel wird bei jeder Veränderung der übergebenen Property *source* die Funktion *subscribe* aufgerufen. Wird die Komponente entladen, also aus der UI entfernt, wird auf die gespeicherte *subscription* die Funktion *unsubscripe* aufgerufen.

Code-Splitting

Damit React-Code ausgeführt werden kann, wird dieser zunächst mittels eines *Module-Bundler* zu einer JavaScript-Datei gebündelt (eine genauere Erläuterung der Funktion eines Module-Bundler ist in Kapitel 2.5 gegeben). Diese Datei kann bei einem großen Projekt jedoch so stark anwachsen, dass dies zu langen Ladezeiten beim Aufrufen der Anwendung führen kann. Dazu bietet JavaScript die Möglichkeit von dynamischen Imports, um Code erst dann zu laden, wenn dieser benötigt wird. Um dies zu nutzen, wird das Schlüsselwort *import* wie eine Funktion aufgerufen. Diese Funktion gibt nach erfolgreichem importieren ein Objekt zurück, welches alle Exporte der importierten Datei beinhaltet.

Statischer Import

```
import { add } from './math';

console.log(add(16, 26));
```

Dynamischer Import

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

Listing 9: Beispiel eines statischen und dynamischen Imports [Reab]

React bietet zusätzlich zu dem dynamischen Import die Möglichkeit, mit der Funktion *lazy* das importierte Objekt als reguläre Komponente zu verwenden. Dabei nimmt die Funktion *lazy* eine weitere Funktion entgegen, welche einen dynamischen Import aufruft. [Reaa]

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

Listing 10: Dynamisches Laden einer Komponente [Reaa]

React-Router

Da durch das Navigieren durch Single-Page-Applications nur der Inhalt eines einzigen HTML-Dokuments neu gerendert wird, muss die URL der Seite nie angepasst werden. Jedoch ist es häufig praktisch, auf eine bestimmte Seite einer Web-Anwendung mittels einer speziellen URL zu gelangen, um diese zum Beispiel als Lesezeichen zu speichern oder als Link weiterzugeben. Auch funktionieren bei solchen Single-Page-Applications die Vor- und Zurück-Buttons der Browser nicht, welche jedoch viele Nutzer gewohnt sind. [vgl. Har20, S. 185]

Um diese Funktionen auch in Single-Page-Applications verfügbar zu machen, gibt es ein Verfahren, welches *Routing* genannt wird. Hierbei werden bestimmte Komponenten abhängig von der im Browser angegebenen URL gerendert. [vgl. Har20, S. 185] Die bekannteste Bibliothek für Routing in React ist *React Router*, welche auch in Audimax verwendet wird. Diese bietet eine große Anzahl an Komponenten und Hooks, woraus ein paar im Folgenden genauer erläutert werden.

Komponenten

React Router stellt mehrere Router-Komponenten bereit, die für unterschiedliche Szenarien genutzt werden können. In diesem Beispiel wird die Komponente *BrowserRouter* genutzt, da es sich hierbei um die Implementierung des Routers für einen Webbrowser handelt. Dieser Router umfasst alle Komponenten, welche auf das Routing zugreifen wollen. Dies ist im Normalfall die gesamte Anwendung.
Innerhalb des Routers kann mittels der *Route*-Komponente angegeben werden, welche Routen – also URLs – es geben soll und bei welcher Route welche Komponente aufgerufen werden soll.

Hooks

Um die URL des Browsers anzupassen, benutzt die Komponente BrowserRouter die HTML-5 History-API, mit welcher der Browserverlauf manipuliert werden kann [Reae]. Um diesen zu manipulieren, kann die Hook *useHistory* genutzt werden. Diese gibt einem Zugriff auf das history-Objekt, mit welchem man zum Beispiel zu einer neuen Seite springen kann. Sollten in der URL Parameter verwendet werden, auf welche man zugreifen möchte, so ist

dies durch die Hook *useParams* möglich. Dazu muss bei der Definition der Route mithilfe eines Doppelpunktes ein Parameter angegeben werden.

```
<Route path="/article/:id" />
```

Listing 11: Aufruf der Route-Komponente

Beim Aufruf der Route */article/1* gibt die Hook useParams ein Objekt mit einem key/value-Paar zurück (*{ id: "1"}*). So lässt sich im Code die in der URL eingegebene Zahl mittels des Keys *id* verwenden.

Redux

In React wird die meiste Logik innerhalb von Komponenten gehalten. Somit werden auch die Zustände (*states*) innerhalb einer Komponente verwaltet. Wie schon in Abschnitt 2.3 erläutert, werden alle Komponenten, welche einen Zustand verwenden, bei Veränderung dessen erneut gerendert. Wird zum Beispiel eine Information wie ein Benutzername in vielen Komponenten der Anwendung benötigt, kann es sinnvoll sein, diesen Zustand global zu verwalten. Dafür kann *Redux* verwendet werden, welches die Zustände in einem globalen Store verwaltet. Für die Verwendung von Redux in React stellt das Redux-Team neben Redux selbst zwei weiter Bibliotheken bereit. Zum einen *React-Redux*, welche eine API für React bereitstellt, um innerhalb von Komponenten mit dem Store zu interagieren, zum anderen *Redux-Toolkit*, welche die Konfiguration eines Stores vereinfacht sowie auch dessen Nutzung.

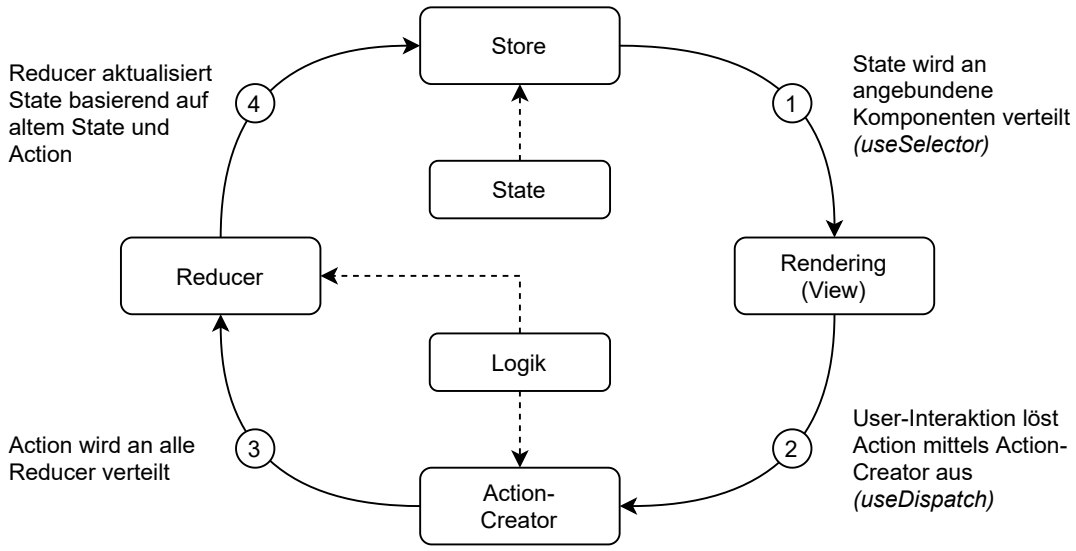

Abbildung 2.2: Darstellung der Funktionsweise eines Redux-Stores[Har20]

Alle Zustände, die global Verfügbar sein sollen, liegen in einem Store. Auf diese Zustände kann innerhalb einer Komponente zugegriffen werden. Zusätzlich kann eine Komponente auch bestimmte Zustände „abonnieren“. Wird ein Zustand abonniert, wird die Komponente, welche diesen abonniert hat, bei jeder Änderung neu gerendert. Hierfür kann die von React-Redux bereitgestellte Hook *useSelector* verwendet werden. Dieser wird eine Funktion übergeben, welche ein Objekt mit den gesamten Zuständen des Stores übergeben bekommt (siehe Listing 12). Diese Funktion wird beim Rendern der Komponente aufgerufen sowie immer dann, wenn sich der Zustand im Store ändert. Um Zustände durch Benutzerinteraktion zu verändern, wird durch eine solche ein Action-Creator aufgerufen. Diesem werden die Informationen übergeben, mit welchen der Zustand geändert werden soll. Der Action-Creator ruft daraufhin den *Reducer* auf, welcher den Zustand im Store manipuliert.

```
const { events } = useSelector((state: State) =>
  state.events
);
```

Listing 12: Zugriff auf die im Redux-Store gespeicherten Events

2.4 Create React App

Eine neue React-Anwendung aufzusetzen ist mit einiger Arbeit verbunden. So müssen alle gewünschten Konfigurationen für zum Beispiel *Webpack* (siehe Abschnitt 2.5) oder TypeScript selbständig vorgenommen werden und auch das Grundgerüst aus einer HTML-Datei sowie der *index.ts* als Einstiegsdatei und der *App.tsx* als React-Hauptkomponente muss selbst implementiert werden. Viele dieser Konfigurationen und auch die Grundstruktur der Dateien sind jedoch bei den meisten Anwendungen identisch und könnten somit automatisiert werden. React selbst empfiehlt für die Erstellung einer Single-Page-Application das Tool *Create React App*. Create React App ist ein CLI-Tool, welches das Aufsetzen einer React-Single-Page-Application erleichtert. Dabei übernimmt Create React App die gesamte Konfiguration unter anderem von Webpack und legt alle benötigten Dateien an. Auch lässt sich beim Anlegen eines Projektes TypeScript als Sprache auswählen. Somit wird auch in diesem Fall die Konfiguration von TypeScript für das Projekt übernommen.

Create React App Configuration Override (CRACO)

Da durch den Einsatz von Create React App viele Konfigurationen übernommen werden und somit die Konfigurationsdateien in den von Create React App genutzten Bibliotheken liegen, können diese nicht überschrieben werden. Es gibt jedoch Fälle in welchen man einige Konfigurationen, welche vorgenommen wurden, überschreiben oder hinzufügen möchte. Für diesen Fall gibt es *Create React App Configuration Override*, kurz CRACO. CRACO

ist ein von GSoft entwickeltes Open-Source Tool, welches durch das Hinzufügen einer Konfigurationsdatei dem Entwickler die Möglichkeit bietet, Konfigurationen wie die von *eslint*, *babel*, *postcss* oder Webpack zu überschreiben. [Gso]

2.5 Module-Bundler

Ein Webbrowser ist in der Lage, JavaScript-Code auszuführen. Besteht eine JavaScript-Datei lediglich aus Funktionsaufrufen der Browser-Api, also aus Funktionen, welche der Browser implementiert hat, kann das Programm ohne Module-Bundler im Browser ausgeführt werden. Jedoch beinhalten übliche Programme nicht nur Aufrufe der Browser-API, sondern beispielsweise auch Aufrufe von Funktionen einer importierten Bibliothek. Zusätzlich bestehen übliche Programme aus mehreren JavaScript- oder auch TypeScript-Dateien. Ähnlich wie TypeScript die Sprache JavaScript erweitert, gibt es auch Erweiterungen der Stylesheet-Sprache *css*, welche nicht von den Browsern unterstützt werden. Auch diese müssen zuerst so übersetzt werden, dass ein Browser diese ausführen kann. Das Ziel eines Module-Bundlers ist es demnach, alle Dateien, welche ein Projekt beinhaltet, zu Dateien zu bündeln, sodass diese vom Browser ausgeführt werden können. Bekannte Module-Bundler sind unter anderem *Webpack*, *Rollup* und *Parcel*.

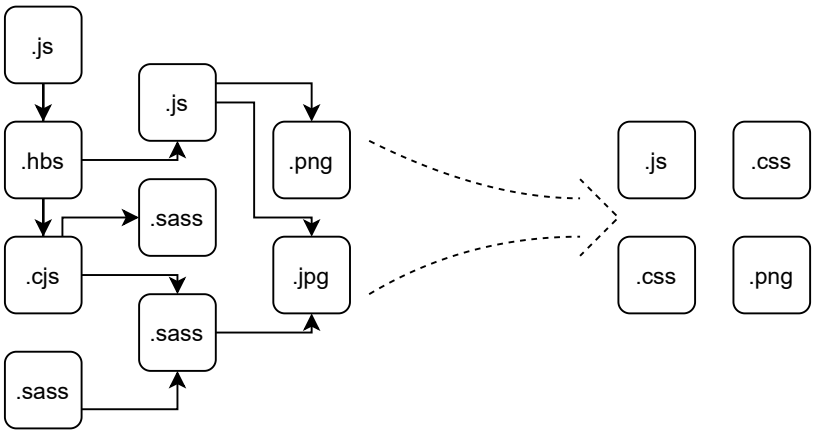

Abbildung 2.3: Darstellung der Webpack-Funktionalität [webb]

3 Konzept

Dieses Kapitel beinhaltet die Konzeptionierung der Modularisierung einer Single-Page-Application. Dazu werden im Folgenden die Grundlagen für erweiterbare Software erläutert sowie der Aufbau einiger Plugin-Systeme in bekannten Anwendungen verglichen.

3.1 Erweiterbare Software

„Eine erweiterbare Anwendung ist eine Anwendung, die es ermöglicht, Funktionen hinzuzufügen, zu ersetzen oder sogar zu entfernen, ohne dass eine Änderung oder ein Zugriff auf den Quellcode erforderlich ist.“ [McV09, i]. McVeigh formuliert dabei die Definition von erweiterbaren Anwendungen sehr allgemein. Dabei geht es nicht darum, Entwicklern die Möglichkeit zu geben, Software an bestimmten stellen zu erweitern, sondern Anwendungen so zu erweitern, dass Funktionen einer Hauptanwendung immer und überall entfernt und verändert werden können. Jedoch lassen sich die von ihm formulierten Ideen [vgl. McV09, S. 4-6] für den in dieser Arbeit beschriebenen Anwendungsfall verwenden, wodurch folgende Anforderungen an eine erweiterbare Software entstehen:

Veränderbarkeit Die Hauptanwendung sollte durch Erweiterungen verändert werden können, sodass diese Funktionalitäten hinzufügen können. Das Löschen sowie Verändern von Funktionalität soll nicht möglich sein, da Erweiterungen keine Veränderungen an der Funktionalität der Hauptanwendung vornehmen sollten.

Unveränderter Quellcode Der Quellcode der Hauptanwendung sollte nicht verändert werden müssen, um eine Erweiterung mit allen gewünschten Funktionen zu erstellen.

Keinen Einfluss Die Änderungen durch Erweiterungen an der Hauptanwendung sollten keinen Einfluss auf die Entwicklung an der Hauptanwendung oder an anderen Erweiterungen haben. Auch sollte die Hauptanwendung losgelöst von allen Erweiterungen fehlerfrei funktionieren.

Erweitern einer Erweiterung Es sollte möglich sein, zusätzlich zu der Hauptanwendung auch schon bestehende Erweiterungen zu erweitern.

Um die Veränderbarkeit der Hauptanwendung zu kontrollieren, werden sogenannte *extension points* benötigt, also Punkte, an welchen die Hauptanwendung erweitert werden darf.

Diese werden von den Entwicklern der Hauptanwendung festgelegt und definieren eine Schnittstelle der Hauptanwendung für die Erweiterungen. Beim Blick auf den Aufbau einer klassischen Webanwendung (siehe Abbildung 3.1) könnten somit die *extention points* *Navigationselemente* und *Hauptinhalt* definiert werden, da es dort möglich sein soll, die Hauptanwendung durch Code einer Erweiterung zu erweitern. Die Seitenleiste soll in diesem Beispiel nicht durch Erweiterungen erweitert werden können, sondern lediglich von der Hauptanwendung genutzt werden und bildet daher keinen *extension point*.

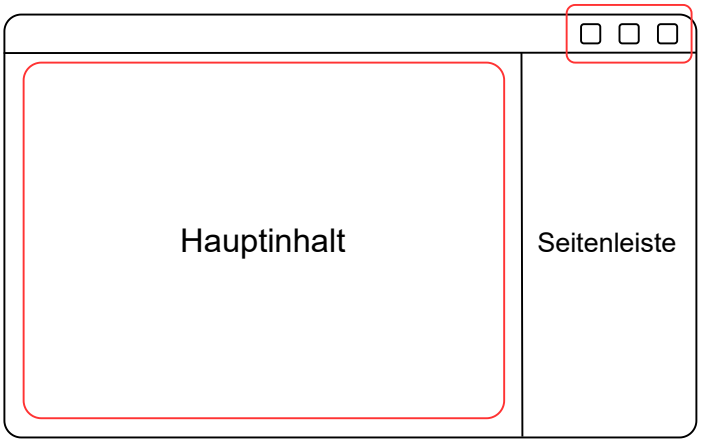

Abbildung 3.1: Mögliche Einstiegspunkte einer klassischen Webanwendung

3.2 Plugin-Systeme

Um ein Konzept für die Modularisierung einer Single-Page-Application zu entwickeln, wurden bereits bestehende modulare Systeme (Plugin-Systeme) betrachtet. Dabei ist lediglich Visual Studio Code eine Single-Page-Application. Dies hat jedoch zunächst keinen Einfluss auf die Konzeptionierung, sondern wird erst bei der Umsetzung interessant. Auf die technischen Besonderheiten der Modularisierung einer Single-Page-Application wird in Abschnitt 3.5 eingegangen. Für die in diesem Abschnitt vorgestellten Systeme wurden die von WordPress, Nextcloud und Visual Studio Code analysiert.

WordPress

WordPress ist eines der bekanntesten freien Content-Management-Systeme. Bekannt geworden ist es als Weblog-System. Dadurch ist WordPress vor allem dafür ausgelegt, bei der Erstellung von Webseiten zu helfen, welche häufig und einfach aktualisiert werden sollen. [vgl. Ste16, S. 9]
Jedoch wird WordPress mittlerweile nicht mehr nur für die Erstellung von Blog-Webseiten verwendet. Durch das umfangreiche Plugin-System und den Open-Source-Gedanken hat jeder die Möglichkeit, Plugins für WordPress zu schreiben, um dieses zu erweitern. So gibt

es Plugins für eine Nutzerverwaltung oder für einen eigenen Online-Shop. Dies zeigt, wie umfangreich das Plugin-System ist, weshalb in dieser Arbeit die Architektur eines Plugins in WordPress genauer betrachtet wird.

Der grundlegende Aufbau eines WordPress-Plugin ist sehr simpel. Dabei wird für ein Plugin lediglich eine php-Datei in dem auf dem Server liegenden Plugin-Ordner benötigt. Diese php-Datei muss einen *Header-Kommentar* beinhalten, welcher die Informationen zu dem Plugin beinhaltet. [vgl. Worb] Dabei reicht es aus, einen Plugin-Namen anzugeben. Sobald dies in der php-Datei definiert ist, wird dieser in der WordPress Oberfläche unter dem Menüpunkt Plugin angezeigt. Neben dem Plugin-Namen können in dem Header-Kommentar weitere Informationen angegeben werden, wie in Listing 13 zu sehen.

```
/**
* Plugin Name:       My Basics Plugin
* Plugin URI:        https://example.com/plugins/the-basics/
* Description:       Handle the basics with this plugin.
* Version:           1.10.3
* Author:            John Smith
* License:           GPL v2 or later
...
*/
```

Listing 13: Auszug aus dem Beispiel Header-Kommentar des WordPress-Plugin Handbuchs [Wora]

Eine wichtige im WordPress-Handbuch definierte Regel lautet: „Don't touch WordPress core“ [Worb]. Um diese Regel einzuhalten und trotzdem eine umfangreiche Erweiterungsmöglichkeit für Plugins zu bieten, stellt WordPress *Hooks* und Funktionen bereit. Wie das Wort Hook schon sagt, bieten diese eine Möglichkeit, Funktionen an vordefinierten Punkten in WordPress einzuhaken. Grundsätzlich unterscheidet WordPress zwischen zwei Arten von Hooks: Zum einen die sogenannten *action*-Hooks und zum anderen die *filter*-Hooks. Um diese Hooks zu verwenden, müssen Funktionen geschrieben werden, welche den gewünschten Code ausführen. Diese Funktionen können dann einer bestimmten Hook zugeordnet werden. Dabei sind action-Hooks dafür da, Daten an einer bestimmten Stelle der Ausführung hinzuzufügen. Zum Beispiel möchte man bei jedem Laden von WordPress einen Request ausführen. Dies wäre mit einer Funktion möglich, welche an die action-Hook *init* gebunden wird. Funktionen, die an action-Hooks gebunden werden, geben nie einen Wert zurück, da es für die Veränderung von Daten die filter-Hooks gibt.
Diese Art von Hooks geben dem Entwickler die Möglichkeit, Daten bei der Ausführung anzupassen. Neben den von WordPress angebotenen Hooks ist es auch möglich, eigene Hooks zu erstellen, um somit Einstiegspunkte für das eigene Plugin zu definieren und dieses durch andere Plugins erweiterbar zu machen.

Neben den Hooks bietet WordPress eine Reihe verschiedener APIs an, um das Hinzufügen von Daten in Wordpress zu erleichtern. Da der Umfang der gesamten WordPress-APIs sehr

groß ist, wird die Funktionsweise dieser an einem Beispiel der Erweiterung der Wordpress-Oberfläche erläutert. In dem folgenden Beispiel soll das Plugin in der Wordpress-Oberfläche einen eigenen Menüpunkt bekommen.
Hierfür stellt WordPress die Funktion *add_menu_page* bereit. Diese Funktion kann nun in einer eigenen Funktion implementiert werden. Dabei werden der Funktion einige benötigte Parameter mitgegeben wie zum Beispiel der Name des Plugins und eine Funktion, welche den Inhalt der Seite zurückgibt.

```
function add_menu() {
  add_menu_page('myPlugin', 'MyPlugin', 'manage_options', 'myplugin',
  ↪  'plugin_content');

  function plugin_content() {
    ?>
        <h1>Mein Plugin</h1>
    <?php
  }
}
```

Listing 14: Funktion zum Hinzufügen eines Menüpunktes in WordPress

Damit diese Menüseite beim Laden der Menüleiste auch angezeigt wird, muss die geschriebene Funktion der passenden Hook übergeben werden. Da es sich hierbei um das Hinzufügen von Daten handelt, wird die Funktion *add_action* verwendet. Dieser Funktion wird die passende Hook übergeben – in diesem Fall *admin_menu* – und als zweiter Parameter die selbst geschriebene Funktion (siehe Listing 15).

```
add_action('admin_menu', 'add_menu');
```

Listing 15: Hinzufügen des Menüpunktes mittels der *admin_menu*-Hook

Nun wird das Plugin als eigener Menüpunkt angezeigt und der Inhalt gerendert. (siehe Abbildung 3.2)

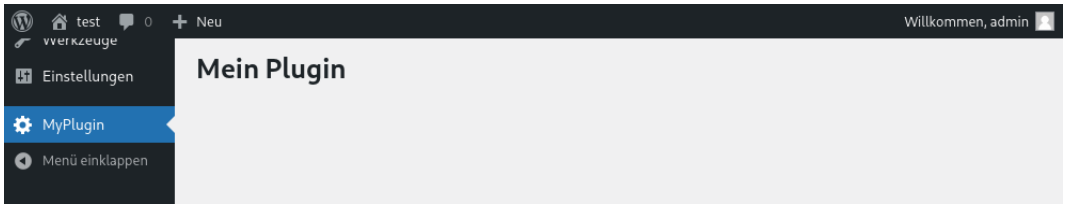

Abbildung 3.2: *MyPlugin* in der WordPress Oberfläche.

Nextcloud

Nextcloud ist eine Open-Source Kollaborationsplattform, die es Nutzern ermöglicht, Erweiterungen zu schreiben, welche das Backend sowie das Frontend erweitern können.

Um eine eigene Nextcloud-Erweiterung zu entwickeln, gibt Nextcloud eine bestimmte Ordnerstruktur vor, wobei nicht alle der in Abbildung 3.3 gezeigten Ordner zwangsläufig benötigt werden.

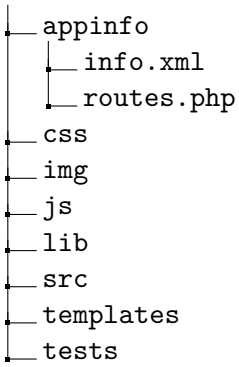

Abbildung 3.3: Ordnerstruktur einer Nextcloud-Erweiterung [Nex]

Im Gegensatz zu einem WordPress-Plugin werden bei einer Erweiterung in Nextcloud die Informationen zu der Erweiterung in eine xml-Datei (*info.xml*) geschrieben. Diese kann dabei deutlich umfangreicher ausfallen, da diese nicht nur Informationen zu der Erweiterung beinhaltet, sondern zusätzlich dort definiert wird, an welchen Stellen Nextcloud erweitert werden soll.

```
<info >
  <id>myplugin</id>
  <name>MyPlugin</name>
  <description>Ths is my custom plugin</description>
  ...
  <navigations>
    <navigation>
      <name>MyPlugin</name>
      <route>myplugin.page.index</route>
      <icon>myPlugin.svg</icon>
      <order>10</order>
    </navigation>
  </navigations>
</info>
```

Listing 16: Auszug der *info.xml*-Datei einer Nextcloud-Erweiterung

So können, wie in Listing 16 zu sehen, mithilfe der xml-Datei Metadaten wie die *id* oder der Name der Erweiterung angegeben werden und zusätzlich auch mittels des xml-Tags *navigation* die Informationen für das Symbol, welches bei Aktivierung der Erweiterung in der Navigationsleiste von Nextcloud erscheinen soll. Hierbei wird der Name, das Icon, eine Ordnungszahl – an dieser Stelle soll das Icon erscheinen – und eine Route angegeben, welche beim Klick auf das Icon aufgerufen werden soll. Damit beim Aufruf der unter dem *route*-Tag angegebenen URL die gewünschte Oberfläche erstellt wird, muss diese Route in der dafür vorgesehen *routes.php* eingetragen werden.

```
return [
    'routes' => [
        ['name' => 'page#index', 'url' => '/', 'verb' => 'GET'],
    ]
];
```

Listing 17: Definition einer Route in der *routes.php*

Die in Listing 17 beschriebene Route definiert, dass bei einem Aufruf der Route */apps/-myapp* die Methode *index* im PageController ausgeführt wird. Unter dem Ordner *lib/Controller* muss daher eine Klasse *PageController* implementiert werden, welche eine Methode *index* beinhaltet, die eine HTML-Seite zurückgibt.

```
class PageController extends Controller
{
    //...
    public function index()
    {
        $response = new TemplateResponse('myapp', 'index');

        return $response;
    }
}
```

Listing 18: Implementierung der Funktion *index*

In dieser Methode kann ein Objekt der Klasse *TemplateResponse* erzeugt werden, indem dem Konstruktor der App-Name sowie der Name des Templates übergeben wird. Mithilfe dieser Informationen kann Nextcloud in der App unter dem Verzeichnis *templates* nachschauen und das passende Template zurückgeben.
Dabei ist die Besonderheit, dass Nextcloud in diesem Fall eine Art Hybrid-Lösung für die Entwicklung der Benutzeroberfläche anbietet. Zwar besteht Nextcloud generell aus mehreren HTML-Dokumenten und stellt daher eine Multi-Page-Application dar; es ist jedoch

möglich, innerhalb solcher Templates Single-Page-Applications zu schreiben. Nextcloud bietet dabei die Möglichkeit, in dem *src*-Ordner eine Vue.js-App zu entwickeln, welche dann in das Template geladen werden kann.

Visual Studio Code

Visual Studio Code ist ein von Microsoft entwickelter Open-Source Code-Editor mit der Möglichkeit, diesen durch eigene Erweiterungen zu erweitern.

Auch Visual Studio Code gibt dabei die Struktur einer Erweiterung vor.

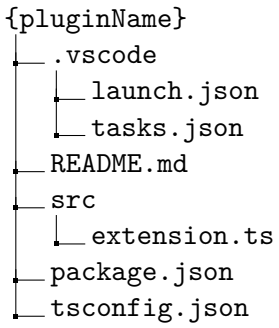

Abbildung 3.4: Ordnerstruktur einer Visual Studio Code Erweiterung [Vis]

Hierbei sind vor allem die beiden Dateien *package.json* und *extension.ts* wichtig. In der package.json werden alle wichtigen Informationen angegeben. Dabei ähnelt die Funktionsweise dieser stark der Funktionsweise der xml-Datei bei Nextcloud.

```
{
  "name": "myapp",
  "displayName": "MyApp",
  ...
  "contributes": {
    "viewsContainers": {
      "activitybar": [
        {
          "id": "myapp",
          "title": "My App",
          "icon": "assets/myapp.svg"
        }
      ]
    }, ...
}
```

Listing 19: Auszug aus der *package.json*-Datei einer Visual Studio Code Erweiterung

In dieser werden ebenfalls Informationen zu der Erweiterung angegeben sowie auch Informationen, an welchen Stellen Visual Studio Code erweitert werden soll. In dem in Listing 19 gezeigten Beispiel wird unter dem *key activitybar* eine ID sowie ein Titel und ein Pfad zu einem Icon angegeben, um ein eigenes Element in der Seitenleiste von Visual Studio Code anzuzeigen.

Um jedoch nicht nur etwas an der Darstellung von Visual Studio Code zu ändern, stellt eine Erweiterung die Datei *extension.ts* zur Verfügung. Diese beinhalten zu Beginn bereits eine Funktionen *activate* und *deactivate*, welche – wie der Name schon sagt – beim Aktivieren oder Deaktivieren der Erweiterung ausgeführt werden. In dieser Datei kann das Modul *vscode* importiert werden, das den Zugriff auf die Extension API von Visual Studio Code ermöglicht. Dieses bietet eine Menge an Funktionen, um Visual Studio Code zu erweitern. Soll beim Klicken eines in der package.json definierten Buttons eine Funktion ausgeführt werden, muss diese Funktion beim Aktivieren der Erweiterung registriert werden. Hierzu kann mittels des importierten vscode-Moduls die Funktion *registerCommand* aufgerufen werden, welcher dann der Name des Commands übergeben wird, sowie eine Funktion, welche beim Aufruf des Commands ausgeführt werden soll.

```
vscode.commands.registerCommand("myapp.myPlugin", () => {
    const panel = vscode.window.createWebviewPanel(
      "myplugin",
      "My Plugin",
      vscode.ViewColumn.One,
      {}
    );

    panel.webview.html = `<h1>My Plugin</h1>`;
  });
```

Listing 20: Registrierung eines Kommandos zum Öffnen eines *WebviewPanel*

In dem in Listing 20 gezeigten Beispiel wird beim Aufruf des Commands *myapp.myPlugin* ein WebPanel erstellt, in welchem eine HTML-Datei mit dem Inhalt „My Plugin“ gerendert wird.

Zusammenfassung

Nach dem genaueren Betrachten der drei Plugin-Systeme lassen sich mehrere Anforderungen an das Konzept der Modularisierung formulieren.

Konfigurationsdatei In jeder der gezeigten Anwendungen ist die Angabe von Informationen zu der Erweiterung essenziell und notwendig, nur in der Art unterscheiden sich die Systeme minimal.

API Wie schon in Abschnitt 3.1 sind für das Kontrollieren der Veränderbarkeit der Hauptanwendung *extension points* zu definieren. Dies geschieht bei den Systemen zum einen durch die Angabe von Informationen in der Konfigurationsdatei, zum anderen aber auch durch eine von der Hauptanwendung bereitgestellten API.

Struktur Durch das Vorgeben einer bestimmten Ordnerstruktur wird dem Entwickler das Erstellen einer Erweiterung erleichtert. Bei Nextcloud wird diese darüber hinaus genutzt, um bestimmte Dateien wie HTML-Templates aus einem Ordner zu nutzen.

Neben den Gemeinsamkeiten der beschriebenen Systeme können auch einzelne Eigenschaften für die Konzeptionierung übernommen werden. Da Visual Studio Code ebenfalls eine mit TypeScript geschriebene Anwendung ist, kann das Konzept, ein Modul aus einer TypeScript-Library zu laden, welches die Funktionen der API bereitstellt, für das zu konzeptionierende System übernommen werden.

3.3 Aufbau

In diesem Abschnitt wird auf Grundlage der gewonnenen Erkenntnisse der Aufbau einer modularen Single-Page-Application skizziert. Eine solche Single-Page-Application besteht im Wesentlichen aus den folgenden drei Komponenten: der Hauptanwendung (*core*), einer Bibliothek (*library*) und den Erweiterungen (*modules*). In der Hauptanwendung werden alle Grundfunktionalitäten sowie die grundlegende Struktur der Benutzeroberfläche implementiert. Sie stellt so eine eigenständige Single-Page-Application dar, welche auch ohne aktivierte Module funktioniert. Auf Grundlage der Struktur der Hauptanwendung werden die Erweiterungspunkte (extension points) für externe Module definiert. Dies geschieht unter der Verwendung der in Abbildung 6 erläuterten Technik, indem alle zu erweiternden Oberflächen des Cores aus einem Objekt (im folgenden *extensionObject* genannt) generiert werden. Aus dem *extensionObject* werden nicht nur die Oberflächen generiert, es dient ebenso den Erweiterungen als eine API, um dem Objekt weitere Elemente hinzuzufügen, welche dann bei Aktivierung eines Moduls gerendert werden können. Nachfolgend wird auf die drei Komponenten detailliert eingegangen.

Core API

Für die Konzeptionierung der Core API lohnt sich ein Blick auf das Konzept der objektorientierten Programmierung. Dort können Module und deren Zusammenhang mittels eines Objektdiagramms dargestellt werden. Die Verallgemeinerung in ein Klassendiagramm bildet dann die Abstraktion für ein erweiterbares modulares System. Die nachfolgenden Absätze konkretisieren diese Ideen.

Die Abstraktion durch ein Klassendiagramm stellt gewissermaßen den Bauplan für Module einer Anwendung dar. Hierzu legen die Klassen fest, welche Attribute und welche Funktionen in einem Objekt der Klasse bereitstehen. Um die Klassen zu erweitern, bietet die

Objektorientierung das Konzept der Vererbung. Dabei werden alle Attribute und Funktionen der Oberklasse ebenfalls für die erbende Klasse bereitgestellt. Es wird somit der Bauplan der Klasse übernommen und durch weitere Attribute und Funktionen wird dieser durch die erbende Klasse erweitert. Der Ansatz, ein Modul der modularen Single-Page-Application als Unterklasse der Hauptanwendung zu definieren, stößt bei der Umsetzung des benötigten Systems auf Probleme. Eine Unterklasse hat die Möglichkeit, die geerbte Oberklasse beliebig zu erweitern, wobei die Oberklasse keine Informationen darüber besitzt, wie diese erweitert wurde. Dies ist jedoch notwendig, da die Hauptanwendung den Code eines Moduls ausführen muss. Zusätzlich soll ein Modul die Hauptanwendung nicht beliebig erweitern können, sondern lediglich über vordefinierte Schnittstellen. Entgegen der Abstraktion von Modulen durch Unterklassen der Hauptanwendung trifft eine Darstellung der *extension points* durch Klassen auf den gewünschten Anwendungsfall zu. Demnach stellt eine Klasse einen Bereich dar, welcher durch ein Modul erweitert werden soll. Durch die Definition einer Schnittstelle (*Interfaces*), welche durch die Klassen implementiert wird, werden die Attribute und Funktionen klar definiert und stellen somit eine Schnittstelle zu der Hauptanwendung bereit. Dies stellt die sogenannte *Core API* dar.

Beispiel

In diesem Beispiel wird auf die zu Beginn des Kapitels skizzierte Webanwendung (siehe Abbildung 3.1) Bezug genommen und die Konzeptionierung der *Core API* am Beispiel des dort beschriebenen *extension point Navigationsleiste* beschrieben.

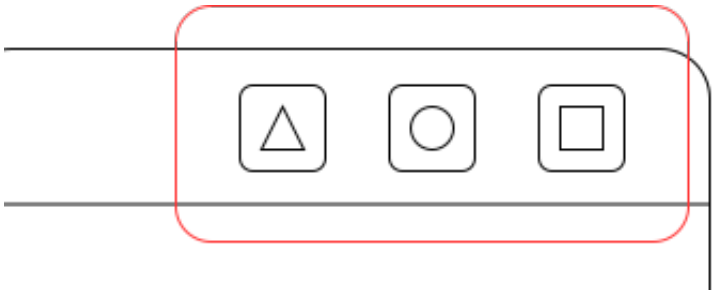

Abbildung 3.5: Darstellung der zu erweiternden Navigationsleiste

In dem in Abbildung 3.5 gezeigten Bereich der Anwendung soll es Modulen möglich sein, der Navigationsleiste ein weiteres Symbol hinzuzufügen. Dabei stellt der rot markierte Bereich einen *extension point* dar. Für diesen wird nun ein Interface definiert.
(Alle folgenden Beispiele sind in TypeScript geschrieben.)

```
export interface INavigationBar {
    navigationSymbols: Symbol[];

    addSymbol(symbol: Symbol): void;
}
```

Listing 21: Interface zur Abstraktion der Navigationsleiste

Das Interface hält ein Array aus Symbolen, aus welchem die Symbole in der Navigationsleiste generiert werden können. Zusätzlich gibt es eine Funktion, die diesem Array ein Symbol hinzufügt und somit als Schnittstelle für Module dient, um die Navigationsleiste um ein solches zu erweitern.

Um den Überblick über alle definierten Klassen zu behalten, wird zusätzlich zu den Interfaces der *extension points* ein weiteres Interface bereitgestellt, welches ein Objekt aller Klassen beinhaltet. Dies stellt somit die Abstraktion des zu Beginn des Abschnitts 3.3 erwähnte *extensionObjects* dar.

```
export interface IExtensionClass {
    navigationBar: INavigationBar;
}
```

Listing 22: Interface zur Abstraktion des extensionObjects

Mittels dieses Objektes kann die Hauptanwendung die Symbole in der Navigationsleiste rendern. Des Weiteren kann eine Erweiterung weitere Symbole zu dem Objekt hinzufügen.

```
{extensionObject.navigationBar.map((navItem) =>
  <NavItem>{navItem}</NavItem>
)}
```

Listing 23: Rendern der Navigationssymbole aus einem Array

```
extensionObject.navigationBar.addSymbol(newSymbol);
```

Listing 24: Hinzufügen eines Navigationsymbols durch ein Modul

Entfernen der Änderungen eines Moduls

Wird ein Modul deaktiviert, sollen alle Änderungen, welche dieses an der Hauptanwendung vorgenommen hat, wieder rückgängig gemacht werden. Diese Aufgabe soll nicht der

Entwickler des Moduls übernehmen, sondern die Hauptanwendung selbst. Dabei gibt das Interface *IExtensionClass* eine Funktion *cleanUp* vor, welche beim Deaktivieren eines Moduls von der Hauptanwendung aufgerufen wird.

Damit sichergestellt wird, dass jeder Funktionsaufruf des Moduls wieder Rückgängig gemacht werden kann, wird beim Ausführen eines Befehls die Funktion in einer Liste gespeichert, welche diesen Befehl wieder rückgängig macht. Die Liste dieser Funktionen wird dabei in einem Objekt zu der dazugehörigen Modul-ID gespeichert. So können beim Entfernen eines bestimmten Moduls mittels dieser ID die passenden Funktionen ausgeführt werden. Eine Implementierung dieses Konzeptes wird in Abschnitt 4.3 an einem Beispiel erläutert.

Bibliothek

Die *Core API* umfasst mehrere Interfaces und Klassen, wobei eine Hauptklasse existiert (*ExtensionClass*), welche die Abstraktion des *extensionObjects* darstellt. Dieses Objekt wird von der Hauptanwendung sowie auch von den Modulen verwendet. Hierbei ist wichtig, dass nicht jedes Modul und die Hauptanwendung eine eigene Instanz der Klasse halten, sondern eine einzige Instanz der *ExtensionClass* zwischen der Hauptanwendung und den Modulen geteilt wird. Daher bietet es sich an, eine Bibliothek zu erstellen, welche die Implementierung sowie die Interfaces der Core API beinhaltet. Dies hat zum einen den Vorteil, dass das *extensionObject* von allen Modulen und der Hauptanwendung aus der Bibliothek statisch geladen werden kann, zum anderen können in dieser Bibliothek zusätzlich React-Komponenten sowie Funktionen und selbstgeschriebene React-Hooks verwaltet werden. Dies reduziert zum einen Code Dopplungen zwischen den Modulen und der Hauptanwendung, zum anderen kann durch die in der Bibliothek definierten React-Komponenten ein einheitliches Design garantiert werden.

3.4 Modul

Aufbau

In diesem Abschnitt wird der Aufbau eines Moduls der modularen Single-Page-Application anhand der zum Ende von Abschnitt 3.2 gegebenen Zusammenfassung erläutert.

Ähnlich zu den Erweiterungen von Visual Studio Code und Nextcloud liegt einem Modul eine vorgegebene Ordnerstruktur zugrunde.

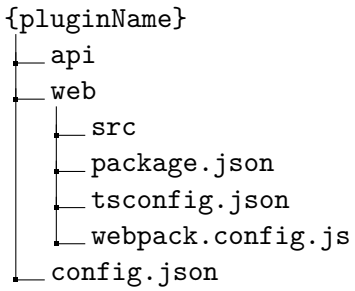

Abbildung 3.6: Ordnerstruktur eines Moduls der Single-Page-Application

Wie in Abbildung 3.6 zu sehen, beinhaltet das Hauptverzeichnis für ein Plugin einen Ordner *api*, welcher die Erweiterung der Backend-Api beinhaltet, einen Ordner *web*, welcher eine eigenständige React-App beinhaltet, sowie ein Konfigurationsdatei *config.json*, welche alle benötigten Informationen zum Plugin beinhaltet. Die config.json sieht dabei wie folgt aus:

```
{
  "id": "myModule",
  "version": "1.0.0",
  "name": "My Module",
  "description": "My first Module",
  "keywords": ["Module", "Extension"]
}
```

Listing 25: Konfigurationsdatei *config.json* eines Moduls

In der Konfigurationsdatei (siehe Listing 25) werden analog zu den *Header Requirements* in WordPress lediglich Metadaten zu dem Modul angegeben. Einstiegspunkte, wie diese in den Konfigurationsdateien von Nextcloud und Visual Studio Code zu finden sind, werden ausschließlich über die Core API definiert.

Der Ordner *src* beinhaltet eine Typescript-Datei nach der vorgegebenen Namenskonvention *{modulName}.tsx*, welche als Einstiegspunkt dient. In dieser Datei muss eine Funktion mit dem Namen *activate* implementiert werden, in welcher mithilfe des *extensionObjects* die Funktionen zur Erweiterung der Hauptanwendung aufgerufen werden können. Neben der benötigten Datei kann zusätzlich eine *routes.ts* implementiert werden. Diese beinhaltet eine Liste an Routen, welche aus einem Pfad bestehen, sowie der Komponente, welche beim Aufrufen des Pfades gerendert wird. Diese Routen werden beim Laden des Moduls der Hauptanwendung hinzugefügt.

```
export const routes: Route[] = [
  {
    path: "/dashboard",
    component: Dashboard,
  },
];
```

Listing 26: Beispiel Inhalt einer *routes.ts*-Datei

Laden der Erweiterung

Da in der Oberfläche alle Module angezeigt werden und dort auch aktiviert und deaktiviert werden sollen, muss das Backend der Hauptanwendung in der Lage sein zu erkennen, welche Module zur Verfügung stehen und welche davon aktiviert sind. Bei aktivierten Modulen muss das Backend zusätzlich den Code für das Frontend bereitstellen.

Hierzu gibt es in der Hauptanwendung einen Ordner *modules*, welcher die Ordner der Module nach dem oben gezeigten Aufbau beinhaltet. Der einzige Unterschied zu diesem ist, dass statt des ganzen Projektes in dem Ordner *web* auf dem Server lediglich die gebündelten JavaScript-Dateien der Erweiterungen liegen. Das Backend geht beim Starten der Anwendung die Unterordner des *modules*-Ordners durch und liest in diesen die *config.json* Datei ein. Alle beinhalteten Informationen werden – solange sie nicht schon vorhanden sind – in der Datenbank abgelegt und können somit mittels eines API-Request vom Frontend abgefragt werden. Zusätzlich zu den Informationen der config.json-Datei wird zu jedem Modul in der Datenbank gespeichert, ob dieses aktiviert ist.

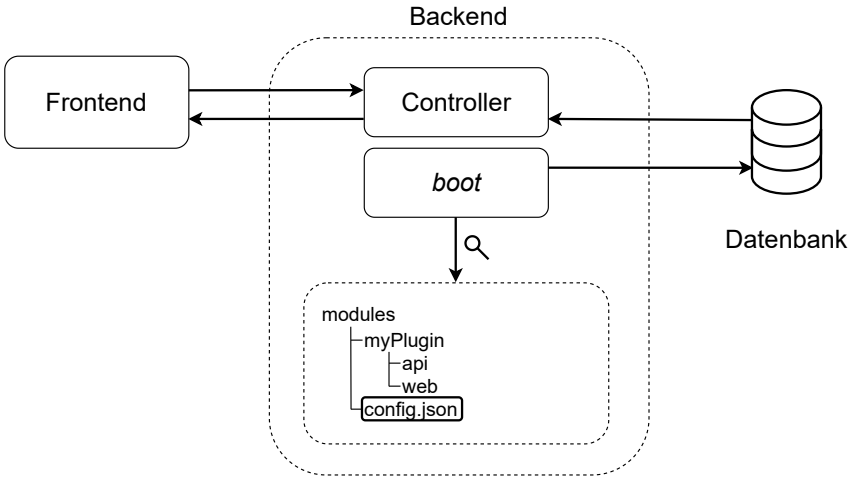

Abbildung 3.7: Laden aller Module im Backend

3.5 Besonderheiten von Single-Page-Applications

Bei den in Abschnitt 3.2 vorgestellten Webanwendungen *Nextcloud* und *WordPress* handelt es sich zwar nicht um Single-Page-Applications, jedoch bieten beide die Möglichkeit, die Erweiterungen als Single-Page-Application zu entwickeln. Dabei gibt jede Erweiterung ein HTML-Dokument zurück, in welches dann dynamisch der Inhalt gerendert wird. Da bei der Modularisierung einer Single-Page-Application jedoch auch die Hauptanwendung aus nur einem HTML-Dokument besteht, fällt die Option für diesen Anwendungsfall heraus. Es wäre zwar möglich, auch innerhalb einer React-Anwendung mittels der Funktion *dangerouslySetInnerHTML* HTML-Code aus dem Backend zu Laden, jedoch bringt dies ein enormes Sicherheitsrisiko mit sich, da dadurch *cross-site-scripting*-Angriffe ermöglicht werden [Reac].

Die Anwendung Visual Studio Code ist eine native Anwendung, welche sich entgegen einer üblichen Webanwendung auf dem Betriebssystem installieren lässt. Da Visual Studio Code das Framework *Electron* nutzt, welches es ermöglicht native Anwendungen mit HTML, JavaScript und CSS zu schreiben, ist Visual Studio Code aus technischer Sicht eine Single-Page-Application. Dabei unterscheidet sich dessen Aufbau jedoch stark von dem einer Single-Page-Application, welche mit React oder Angular geschrieben wurden und welche in dieser Arbeit behandelt werden.

Um eine Single-Page-Application zu modularisieren, muss es möglich sein, dynamisch eine gebündelte JavaScript-Datei eines Moduls vom Server in der Anwendung zu laden. Zwar gibt es, wie in Abschnitt 2.3 erläutert, dynamische Imports in JavaScript, diese können jedoch nur den Code dynamisch laden, indem dieser beim Prozess des Bündels verfügbar ist, erst danach aufgeteilt wird und somit zur Laufzeit nachgeladen wird. Das liegt vor allem an den Module-Bundlern, da diese beim Prozess des Bündels das *Tree Shaking* Konzept verwenden. [Weba] Mittels dieses Konzeptes soll der Code, welcher nicht verwendet wird, aus der gebündelten Datei entfernt werden. Da der Code eines dynamischen Imports von einem Server beim Prozess des Bündels nicht erreichbar ist und diese Zeile somit als „toter“ Code angesehen wird, kann so keine gebündelte JavaScript-Datei dynamisch in einer Single-Page-Application geladen werden.

Aushilfe dabei bietet das mit Webpack 5 veröffentlichte Feature *Module Federation*, welches im folgenden Abschnitt erläutert wird.

3.6 Module Federation

Module Federation ist ein Feature für den Webpack-Bundler, welches seit dem 10.10.2020 mit dem Release von Webpack 5 zur Verfügung steht. Das Ziel von Module Federation ist es, eine neue Methode für die gemeinsame Nutzung von Code zwischen mehreren Anwendungen zu bieten [vgl. JH20, S. 6]. Mittels Module Federation ist es möglich, zwischen zwei Anwendungen Code zu teilen, indem eine Anwendung ihren Code *exposed* (freigibt) und

eine zweite Anwendung diesen *consumed* (konsumiert). Wichtig hierbei ist, dass dieser Code zur Laufzeit konsumiert werden kann. Um dies zu ermöglichen, bündelt Webpack eine JavaScript-Manifest-Datei, welche die Informationen zu den freigegebenen und geteilten Modulen beinhaltet. Wird diese Datei im Browser geladen, registriert dieser eine globale Variable, welche eine *get*- und eine *override*-Funktion enthält. Über dieses in der Variablen gespeicherte Objekt kann der Code der Anwendung geladen werden. [vgl. JH20, S. 37-38] Werden zwei Anwendungen getrennt voneinander entwickelt, wobei die erste Anwendung *App1* die zweite Anwendung *App2* einbinden möchte, ist es durch Module Federation möglich, den Code von *App2* in *App1* zu laden. Dabei wird immer die aktuellste Version von *App2* verwendet, da der Code zur Laufzeit konsumiert wird. Geladen werden kann der Code dabei auf zwei Arten. Zum einen kann mittels der *get*-Funktion der Code dynamisch geladen werden (siehe Listing 27). Dabei muss in der Module Federation Konfiguration von App1 kein Konsument angegeben werden. Zum anderen ist es auch möglich, bei Angabe eines Konsumenten in der Konfiguration den Code über einen gewöhnlichen asynchronen Import zu laden (siehe Listing 28).

```
window.app2.get('App2Module')
```

Listing 27: Laden des Codes mittels der get-Funktion

```
import("app2/App2Module")
```

Listing 28: Laden des Codes mittels eines asynchronen Imports

Dieser Ansatz von Module Federation ähnelt sehr dem Ansatz von Micro-Frontend-Frameworks. Hierbei gibt es jedoch einen wesentlichen Unterschied: Module Federation dient hauptsächlich dem Teilen von Code zwischen verschiedenen Anwendungen, während es bei Micro-Frontend-Frameworks um die gemeinsame Nutzung der Benutzeroberfläche geht [vgl. JH20, S. 12].

Geteilte Bibliotheken

Bei der Entwicklung einer üblichen Single-Page-Application wird häufig auf JavaScript Bibliotheken zurückgegriffen, um bestimmte Funktionalitäten nicht eigenständig implementieren zu müssen. Bei der Nutzung von Module Federation kann der Fall auftreten, dass zwei Anwendungen die gleichen Bibliotheken benutzen. Dies kann dabei zu kritischen Problemen führen. [vgl. JH20, S. 93]

Duplikationen Werden die Bibliotheken zwischen den Anwendungen nicht geteilt, so kann es zu längeren Ladezeiten der Anwendung kommen.

Verschiedene Versionen Wird die Version beim Teilen der Bibliotheken nicht korrekt gesetzt, so kann dies zum Abstürzen der Anwendung kommen, da eine der beiden Anwendungen eine falsche Version verwendet.

Singletons Bei gewissen Bibliotheken ist es notwendig, dass sich Anwendungen genau eine Instanz dieser teilen.

Module Federation bietet bei der Konfiguration verschiedene Möglichkeiten, um diesen Problemen vorzubeugen. Diese werden zum Ende von Abschnitt 4.2 genauer erläutert.

4 Implementierung

In diesem Kapitel wird die Implementierung der modularen Single-Page-Application erläutert. Hierbei wird die konkrete Umsetzung an der in Kapitel 1.2 beschriebenen Kommunikationsplattform *Audimax* beschrieben.

4.1 Herangehensweise

Da die Anwendung *Audimax* zu Beginn der Entwicklung nicht nach dem in Abschnitt 3.3 beschriebenen Konzept entwickelt wurde, ist die in diesem Kapitel beschriebene Implementierung keine von Grund auf neue Anwendung, sondern eine Weiterentwicklung eines bestehenden Systems. Aus diesem Grund muss eine Herangehensweise für das Integrieren des Konzeptes in Audimax festgelegt werden.
Das Frontend von Audimax besteht aus einer einzigen Single-Page-Application. Um nun das zuvor beschriebene Konzept umzusetzen, wird die aktuelle Anwendung als *Core* angesehen. Für die Implementierung sollen nacheinander die Funktionen in eigene Module ausgelagert werden. Dabei wurde mit der Auslagerung der Konferenz in ein eigenes Modul begonnen. Aus diesem Grund wird die in diesem Kapitel beschriebene Implementierung diesen Prozess beschreiben.

4.2 Konfiguration

Library

Für die Entwicklung einer eigenen lokalen React-Library kann wie bei dem Core Create React App verwendet werden. Jedoch gibt es das Tool *create-react-library*, welches unter anderem Create React App verwendet, allerdings anstatt des Module-Bundlers Webpack *Rollup* verwendet. Rollup wurde für das Entwickeln von effizienten JavaScript-Bibliotheken entwickelt [Har17]. Dieses Tool kann ähnlich wie Create React App in der Konsole ausgeführt werden und erstellt nach Angaben einiger Parameter einen Ordner mit allen benötigten Konfigurationen. In dem Core sowie auch in den Modulen kann diese Bibliothek mithilfe des Node Package Managers von einem lokalen Pfad aus installiert werden.

```
npm install <pfad_zur-bibliothek>
```

Listing 29: Installation der lokalen Bibliothek

Nach der Installation erscheint die Bibliothek als *dependecy* in der *package.json* und kann somit von den Modulen oder dem Core verwendet werden.

Core

Zu Beginn der Entwicklung von Audimax wurde die Anwendung mit dem Tool *Create React App* aufgesetzt. Die aktuelle Version dieses Tools beinhaltet jedoch nur Webpack 4 und somit noch keine Unterstützung von Module Federation. Allerdings bietet Create React App einen pre-Release mit der Version *5.0.0-next.47* (Stand: 26.11.2021) bei dem Node package manager *npm* an, welche Webpack 5 unterstützt. Ein integrierter Support für Module Federation ist hingegen noch in Bearbeitung und wurde daher noch nicht offiziell veröffentlicht[Aya]. Um trotzdem dieses Feature nutzen zu können, kann das package *craco-module-federation* genutzt werden. Hierbei ist zu erwähnen, dass auch dieses Plugin aktuell noch nicht für den Produktiveinsatz bestimmt ist. Hierbei fügt *craco-module-federation* das Modul *craco* dem Projekt hinzu, sodass in der Konfigurationsdatei *craco.config.js* eine Webpack-Konfiguration angegeben werden kann. In dieser Datei wird als Webpack Plugin das zuvor genannte Plugin *craco-module-federation* hinzugefügt.

```
const cracoModuleFederation = require('craco-module-federation');

module.exports = {
    plugins: [{
        plugin: cracoModuleFederation,
      },
    ]
}
```

Listing 30: Hinzufügen des Webpack-Plugins *craco-module-federation* in der *craco.config.js*

Um die Konfigurationen für Module Federation vorzunehmen, muss neben der Konfigurationsdatei *craco.config.js* zusätzlich eine *modulefederation.config.js* Datei hinzugefügt werden. In dieser werden die Konfigurationen vorgenommen, auf welche das *craco-module-federation*-Plugin zugreift.

Hierbei wird, wie in Listing 31 zu sehen, zunächst der Name gesetzt. Danach werden Module angegeben, welche vom Core den Modulen zur Verfügung gestellt werden. Dazu wird zunächst der öffentliche Pfad angegeben, unter welchem die Module erreichbar sein sollen, sowie der Pfad zu den freizugebenden Modulen. In diesem Fall liegen diese im Core unter dem Pfad */src/lib/audimax/components*. Diese Module müssen vom Core zur Verfügung gestellt werden, da in diesen auf Funktionen des Cores zugegriffen wird und sie daher nicht ohne Weiteres in die Library ausgelagert werden können. Des Weiteren wird der Dateiname der Datei angegeben, in welche Webpack das Programm bündelt. Diese Datei stellt die in Abschnitt 3.6 erwähnte JavaScript-Manifest-Datei dar. Zusätzlich werden auch geteilte Dependencies angegeben. Auf diesen Punkt wird am Ende dieses Kapitels eingegangen.

```
module.exports = {
  name: "core",
  exposes: {
    "./components": "./src/lib/audimax/components",
  },
  filename: "core.js",
  shared: { /* dependencies */  }
};
```

Listing 31: Konfiguration des Module Federation Plugins

Für die Verwendung von *Module Federation* sollte die Anwendung „gebootstrapped“ werden. Hierbei bedeutet *bootstrapping*, dass der Einstiegspunkt der Anwendung, also die Datei von welcher aus Webpack beginnt die Dateien zu bündeln, lediglich die Hauptanwendung asynchron laden sollte. Dies schützt bei der Ausführung der Anwendung vor möglichen *race conditions*, da Webpack so die Möglichkeit besitzt, zuerst den Rest der Importe der Anwendung zu verarbeiten, bevor die Komponenten der Anwendung ausgeführt werden. Wäre dies nicht der Fall und der Einstiegspunkt wäre eine JavaScript-Datei mit dem Inhalt aus Listing 32, so würde der Code direkt ausgeführt und Webpack hätte keine Zeit, die externe Komponente *Header* zu laden. [vgl. JH20, S. 46-47]

```
import ReactDOM from "react-dom";
import Header from "nav/Header";
ReactDOM.render(<Header />, document.getElementById("app"));
```

Listing 32: Beispiel: Race Condition [JH20]

Mit der Veröffentlichung von Webpack 5 wurde das automatische Hinzufügen sogenannter *Polyfills* beendet. Diese wurden zuvor automatisch den benötigten Paketen hinzugefügt, um für Node.js entwickelte Module auch für die Frontend-Entwicklung und somit für das Laufen in einem Browser verfügbar zu machen. Da mittlerweile jedoch viele Packages ausschließlich für die Verwendung im Frontend und somit im Browser geschrieben werden, wird das automatische Hinzufügen der vielen Polyfills von Webpack als unnötig angesehen [Web20].
Audimax selber verwendet einige Packages, welche in ihren abhängigen Modulen weitere Module beinhalten, die nicht ohne diese Polyfills auskommen. Um nicht selber diese umzuschreiben oder auszutauschen, gibt es die Möglichkeit, innerhalb der Webpack-Konfiguration für diese Module manuell einen Polyfill hinzuzufügen. Hierzu kann wieder von der Verwendung von CRACO profitiert werden, da dies die Überschreibung der Webpack Konfiguration vereinfacht.

```
module.exports = {
  webpack: {
    configure: {
      resolve: {
        fallback: {
          path: require.resolve("path-browserify"),
          ...
};
```

Listing 33: Konfiguration eines Polyfills

Da das in Listing 33 stehende Module *path* ein Modul ist, welches nicht benötigt wird, sollte das Programm in Node.js laufen, muss hierfür das package *path-browserify* als Fallback angegeben werden, da innerhalb eines Browser *path* nicht verfügbar ist und somit dieser Polyfill benötigt wird.

Module

Für die einzelnen Module ist es nicht notwendig, eine React-App mittels Create React App zu erstellen, da diese Module nie als eigene Anwendung laufen sollen. Zusätzlich sollen in Zukunft einem Entwickler die Erstellung eines Moduls und somit die Konfiguration deutlich erleichtert werden, sodass langfristig ein eigenes Tool die folgend beschriebenen Konfigurationen übernehmen soll.

Wichtig ist bei der Konfiguration des Moduls hauptsächlich die *webpack.config.js*-Datei.

```
module.exports = {
 plugins: [
   new ModuleFederationPlugin({
     name: "conference",
     filename: "bundle.js",
     exposes: {
       "./conference": "./src/conference",
       "./routes": "./src/routes",
     },
     shared: {
       ...deps,
     },
   }),
 ],
};
```

Listing 34: Auszug aus der Webpack-Konfiguration des Moduls *conference*

Die *webpack.config.js*-Datei ähnelt der der Hauptanwendung. Wie in Abschnitt 3.4 erläutert folgt die Einstiegsdatei der Namenskonvention *{modulName}.tsx*. Da in dem in Listing 34 gezeigten Beispiel das Modul *conference* heißt, wird dieser Name ebenfalls als *name* im *ModuleFederationPlugin* gesetzt. Zusätzlich wird die Datei unter dem key *exposes* freigegeben. Neben der Einstiegsdatei des Moduls wird zusätzlich auch die *routes*-Datei freigegeben, da mittels dieser, wie in Abschnitt 3.4 erläutert, die benötigten Routen des Moduls in der Hauptanwendung geladen werden.

Geteilte Bibliotheken

Die Konfiguration der geteilten Bibliotheken ist für die Hauptanwendung sowie für alle Module identisch. Daher wird diese im Folgenden gesondert erläutert.

Module Federation bietet für geteilte Bibliotheken mehrere Konfigurationsmöglichkeiten, um den in Abschnitt 3.6 gezeigten Problemen vorzubeugen. Dabei beschränkt sich die benötigte Konfiguration in dieser Anwendung auf die Optionen *singleton* und *requiredVersion*. Die Konfiguration kann hierbei gut an der lokalen Library erläutert werden, da diese zwischen Modulen und dem Core geteilt werden muss, damit die Modularisierung funktioniert.

```
"@audimax/lib": {
  singleton: true,
  requiredVersion: require("../lib/package.json").version,
},
```

Listing 35: Konfiguration der geteilten Bibliotheken

Mittels der Option *singleton* wird für das gegebene Package angegeben, ob nur eine Instanz dessen ausgeführt wird [vgl. JH20, S. 98]. Dies ist für die Implementierung der Modularisierung notwendig, da andernfalls der Core sowie ein Modul eine eigene Instanz des extensionObjects halten. Das führt dazu, dass beim Hinzufügen von Elementen durch ein Modul die Änderung nur an der eigenen Instanz des extensionObjects vorgenommen wird. Dem Core würden dadurch die Änderungen in seiner Instanz nicht vorliegen und somit diese in der Oberfläche nicht erscheinen. Als Zweites wird für das Package unter der Option *requiredVersion* eine benötigte Version angegeben. In Listing 35 wird dafür auf die in der package.json stehenden Version zugegriffen. Wird nun zum Beispiel ein Modul nachgeladen, welches jedoch eine höhere Version der geteilten Bibliothek benötigt, so lädt es die eigene verwendete Bibliothek nach.

4.3 Library

extensionObject

Interfaces

Die Abstraktion des *extensionObjects* wurde in Abschnitt 3.3 durch das Interface *IExtensionClass* dargestellt. Für die tatsächliche Implementierung stellt dies nun das Interface *IAudimax* dar. Wie zu Beginn erwähnt erfolgt die Implementierung des Konzeptes an der Erstellung eines Moduls Konferenz. Um die extension points zu definieren, muss zunächst ermittelt werden, wo die Konferenz in der Oberfläche auftaucht. Dies geschieht zum einen als Icon im Header, um dann eine private Konferenz zu erstellen, zum anderen auch als eigener Kanal innerhalb von Kursen. Dementsprechend müssen alle Kanaltypen sowie die Symbole im Header aus dem Objekt generiert werden. Das Interface sieht demnach wie folgt aus:

```
export interface IAudimax {
  header: Header;
  channel: Channel;

  cleanUp(moduleId: string): void;
}
```

Listing 36: Interface IAudimax

Zusätzlich wird die in Abschnitt 3.3 beschriebene Funktion *cleanUp* benötigt, die das Hinzufügen der Elemente durch ein Modul rückgängig macht.

Der Aufbau des Objektes, aus welchem der Header gerendert werden soll, wird in dem Interface *IHeader* beschrieben. Es beinhaltet als Attribute eine Liste aus Objekten des Interfaces *HeaderAction*, aus welchem die Icons in der Navigationsleiste gerendert werden. Ebenfalls enthält es das Objekt, das eine Liste an Funktionen zu einer gegebenen *moduleId* speichert, die es benötigt, um das Hinzufügen von Funktionen rückgängig zu machen. Als Funktionen beinhaltet das Interface die Funktion *addAction* und die Funktion *removeAction*. Diese Funktionen fügen der Liste *headerActions* ein oder mehrere Elemente hinzu oder entfernen diese.

Das zweite Objekt, welches das Interface *IAudimax* bereitstellt, ist ein Objekt, aus dem die Kanaltypen gerendert werden. Dazu stellt das Interface *IChannel* drei Attribute bereit:

channelFeatures: string[] Eine Liste an Zeichenketten, welche alle IDs der einzelnen Funktionalitäten von Audimax beinhaltet (zum Beispiel: *calendar*, *conference*).

channelOptions: SelectOption[] Eine Liste von Objekten, aus welchen die Auswahl des Kanals beim Erstellen eines solchen gerendert wird.

channelTypeArgs: ChannelTypeArgsType Ein Objekt, welches für jeden *key* eines Kanaltyps den Inhalt des Kanals sowie das Icon beinhaltet.

Zu jedem der drei Attribute gibt es jeweils eine Funktion, welche dem Objekt ein weiteres Element hinzufügt und eine Funktion, welche das Hinzufügen rückgängig macht.

Klassen

Die Klassen *Header* und *Channel* implementieren das dazugehörige Interface, deklarieren die benötigten Attribute und initialisieren diese mit einem leeren Objekt oder einer leeren Liste. Zusätzlich besitzen beide Klassen ein Objekt aus key/value-Paaren, in welchem zu der ID eines Moduls die Funktionen in einer Liste gespeichert werden, welche die durchgeführten Änderungen wieder Rückgängig machen. Im Folgenden wird dies Beispielhaft an der Implementierung der Header-Klasse gezeigt.

Beispiel

```
removeActions(headerActions: HeaderAction[]) {
  return function (header: Header) {
    header.headerActions = header.headerActions.filter(
      (h) => !headerActions.includes(h)
    );
  };
}
```

Listing 37: Implementierung der removeAction-Funktion

In der Klasse wird eine Funktion *removeAction* implementiert, welche wiederum eine weitere Funktion zurückgibt, die ein Objekt der Header-Klasse übergeben bekommt und aus diesem die Elemente herausfiltert, welche nicht der Funktion headerActions übergeben werden.

```
addActions(headerActions: HeaderAction[], moduleId: string) {
  this.headerActions = this.headerActions.concat(headerActions);
  ...
  this.memoryCalls[moduleId].push(this.removeActions(headerActions));
}
```

Listing 38: Implementierung der addAction-Funktion

In der Funktion *addAction* wird zuerst die übergebene Liste an headerActions, also diese, welche durch das Modul der Hauptanwendung hinzugefügt werden soll, der bestehenden

Liste hinzugefügt. Daraufhin wird in der Variablen *memoryCalls* zu dem key – der Modul-ID – die passende Funktion übergeben, welche das Hinzufügen rückgängig macht. In diesem Fall ist dies die *removeAction* Funktion. Da dieser ebenfalls die hinzuzufügende Liste des Typs *HeaderAction* übergeben wird, kann beim Ausführen der gespeicherten Funktion diese wieder entfernt werden.
Das Ausführen der Funktionen übernimmt die in der Klasse *Audimax* implementierte Funktion *cleanUp*. Diese wird mir der zu deaktivierenden Module-ID aufgerufen und kann somit über alle für diesen Key abgespeicherten Funktionen iterieren und diese ausführen (siehe Listing 39).

```
cleanUp(moduleId: string) {
  const cleanUpHeaderFunctions = this.header.memoryCalls[moduleId];
  //...
  return;
  cleanUpHeaderFunctions.forEach((fn) => {
    fn(this.header);
  });
}
```

Listing 39: Implementierung der cleanUp-Funktion

Damit letztendlich alle Module sowie der Core auf das extensionObject zugreifen können, wird in der *index.tsx* ein Objekt der Klasse Audimax erzeugt und dieses exportiert.

```
export const audimax = new Audimax();
```

Listing 40: Anlegen und Exportieren des extensionObjects *audimax*

4.4 Core

Über die Oberfläche des Cores muss es durch die Modularisierung möglich sein, Module und deren Code zu laden. Zusätzlich muss dieser so angepasst werden, dass alle zuvor definierten extension points mittels des extensionObjects gerendert werden. Letzteres ist dank React leicht umzusetzen. Hierbei muss lediglich das von der Library exportierte Objekt *audimax* importiert werden, um dann auf die Liste headerActions zuzugreifen und aus dieser die Elemente der Navigationsleiste zu rendern.

```
import { audimax } from "@audimax/lib";
//...
{audimax.header.headerActions.map((a: HeaderAction) => (
  <HeaderAction action={a} />
))}
//...
```

Listing 41: Rendern der Navigationselemente mittels des audimax-Objektes

Das Laden der aktivierten Module ist hingegen umfangreicher. Zunächst muss sichergestellt werden, dass das Modul nur einmalig zum Laden oder Neuladen der Seite geladen und die im Modul implementierte Funktion *activate* ausgeführt wird. Wäre dies nicht der Fall und dies würde häufiger geschehen, so würde zum Beispiel die Funktion *addAction* mehrmals ausgeführt werden und das Icon des Moduls somit mehrfach in der Navigationsleiste angezeigt werden. Für das einmalige Laden der Module ist die Komponente *ModuleProvider* verantwortlich.

ModuleProvider

Der ModuleProvider hat lediglich die Aufgabe des Ladens der aktivierten Module und soll dabei nur einmal beim Laden der Anwendung oder bei Aktivierung eines Moduls über die Oberfläche gerendert werden. Um dies zu gewährleisten umschließt die Komponente die von React-Router bereitgestellte Komponente *BrowserRouter* (Abschnitt 2.3). Damit geschieht das gesamte Routing und somit die gesamte Interaktion innerhalb des ModuleProviders und dieser wird nicht durch das Verändern der Benutzeroberfläche neu gerendert.

```
export const App = () => {
  return (
    <Provider store={store}>
      <ModuleProvider>
        <BrowserRouter>
          ...
        </BrowserRouter>
      </ModuleProvider>
    </Provider>
  );
};
```

Listing 42: Aufruf des ModuleProviders in der Anwendung

Wie in Listing 42 zu sehen gibt es jedoch eine Komponente, die eine Ebene über dem ModuleProvider steht. Dies ist der Provider von React-Redux. Dieser muss sich eine Ebene

über dem ModuleProvider befinden, da die Information, welche Module aktiviert sind, im Redux-Store liegt und der ModuleProvider auf diesen Zustand mittels der Hook useSelector zugreift.

```
const { modules } = useSelector(
  (state: RootState) => state.module
);
```

Listing 43: Laden der Module aus dem Redux-Store

Das Laden des Codes der Module geschieht in einer Funktion, welche der Hook useEffect übergeben wird. Dadurch kann das Rendern der Komponente kontrolliert werden. Diese Funktion iteriert über alle aktivierten Module und ruft die Funktion *activateModule* auf, welcher das Modul übergeben wird.

Aktivieren eines Moduls

Um den Code eines Moduls zu laden, wird zunächst der HTML-Script-Tag dynamisch in den HTML-Code geladen. Dies geschieht durch einfaches JavaScript. Dazu wird ein Script-Element erzeugt, dessen Attribut *src* die URL der gebündelten JavaScript-Datei des Moduls zugewiesen wird und daraufhin den Metadaten des HTML-Dokuments hinzugefügt wird. Ist der Skript-Tag erfolgreich geladen, wird von Module Federation eine globale Variable im Browser mit den in Abschnitt 3.6 erläuterten Methoden *get* und *override* spezifiziert. Somit kann nun in der Funktion *loadComponent* das Modul geladen werden. Dazu wird dieser Funktion der *scope* (der Name der globalen Variablen) sowie das Modul (in diesem Fall die Einstiegsdatei des Moduls *conference.tsx*) übergeben.

```
export async function loadComponent(scope: string, module: string) {
  //...
  const factory = await window[scope].get(module);
  const Module = factory();
  return Module;
}
```

Listing 44: Implementierung des Laden eines Moduls

Da bei der Implementierung von Modulen das Implementieren einer *activate*-Funktion vorgegeben ist, kann diese mit dem geladenen Modul aufgerufen werden. Dabei werden die vom Modul aufgerufenen Funktionen der Core API ausgeführt und somit das extensionObjekt um die Elemente erweitert, welche das Modul vorsieht. Zusätzlich ist es auch möglich, den Core mittels einer *routes.tsx* um weitere Routen zu erweitern. Hierbei wird ein zweites Mal die loadComponent-Funktion aufgerufen. Der scope ändert sich dabei nicht, jedoch

das Modul, da die Routen nicht aus der *conference.tsx*, sondern aus der *routes.tsx* geladen werden. Das durch die Funktion zurückgegebene Modul beinhaltet nun ein Objekt namens *routes*. Dieses kann den Routen des Cores hinzugefügt werden.
Da das Laden der einzelnen Module asynchron ist, lädt der ModuleProvider seine *children* direkt und wartet demnach nicht, bis die Elemente der Module dem extensionObject hinzugefügt wurden. Da das extensionObject nur ein importiertes Objekt ist, besitzt dieses nicht die Eigenschaften eines Zustandes und löst somit bei einer Veränderung auch kein erneutes Rendern der Komponente aus. Aus diesem Grund muss der ModuleProvider garantieren, dass seine children erst nach dem Laden der Module gerendert werden.
Um dies zu gewährleisten, hält der ModuleProvider einen Zustand, der initial auf *false* gesetzt ist. Dieser wird erst nach dem Laden der Module auf *true* gesetzt und löst somit ein neues Rendern der Komponente aus. Für die bessere Benutzererfahrung wird, solange die Module nicht geladen sind, ein Ladekreis gerendert, und erst, wenn der Zustand *true* ist, die children und somit der Inhalt der Anwendung gerendert.

Da das extensionObject in der Library liegt und somit leer initialisiert wird, fehlen dem Objekt zu Beginn auch die Elemente, welche der Core an den extension points anzeigen möchte. Diese sollen ebenfalls lediglich beim Laden der Anwendung dem Objekt einmalig hinzugefügt werden. Dafür ist der *LoadCoreProvider* verantwortlich.

LoadCoreProvider

Der LoadCoreProvider liegt, wie auch der ModuleProvider, außerhalb des BrowserRouters. Dabei stellt der LoadCoreProvider sicher, dass die Elemente des Cores beim Laden der Anwendung einmalig dem extensionObject hinzugefügt werden. Hierbei wird die useEffect-Hook verwendet, da die der Hook übergebene Funktion somit beim erstmaligen Rendern der Komponente ausgeführt wird. Zurück gibt die Komponente lediglich ihre *children*, also alle Komponenten, die sie umschließt.

```
useEffect(() => {
  audimax.header.addActions(actions);
    ...
  }, []);

return <>{children}</>;
```

Listing 45: Aufruf der Core API im LoadCoreProvider

4.5 Module

Für die Entwicklung eines Moduls muss in der Einstiegsdatei, welche in diesem Fall die *conference.tsx* ist, die Funktion *activate* implementiert werden. Innerhalb dieser Funktion können Funktionen der Core API, also des extensionObjects *audimax* aufgerufen werden.

Um ein eigenes Icon in der Navigationsleiste von Audimax anzuzeigen, kann demnach die Funktion *addAction* aufgerufen werden. Dieser wird ein Array aus HeaderActions übergeben sowie die Modul-ID.

```
audimax.header.addActions(
  [
    {
      displayName: "Konferenz",
      icon: <VideoCallIcon />,
      to: "/meeting/create",
    },
  ],
  "conference"
);
```

Listing 46: Aufruf der Core API Funktion addAction

In dem Objekt des Typs HeaderAction wird ein *displayName* angegeben, eine Komponente, welche das Icon beinhaltet sowie ein Pfad, zu dem navigiert werden soll, wenn auf das Icon geklickt wird.

Abbildung 4.1: Durch das Konferenz-Modul hinzugefügtes Icon

Auf diese Weise kann jede der in Abschnitt 4.3 erwähnte Funktion von einem Modul aufgerufen werden und somit der Core erweitert werden.

Bei dem in Listing 46 gezeigten Funktionsaufruf wird der Funktion ein Pfad übergeben. Jedoch weiß der Core noch nicht, welche Komponente nach dem Aufrufen des Pfades gerendert werden soll. Dazu muss dieser Pfad mit der dazugehörigen Komponente in der *routes.tsx* angegeben werden.

```
export const routes: Route[] = [
  {
    path: "/meeting/create",
    component: CreateMeeting,
  },
];
```

Listing 47: Auszug aus der routes.tsx-Datei

Dabei wird in der Liste an Routen ein Element angegeben, welches den Pfad angibt und die dazugehörige Komponente.

Somit ist es möglich, der Navigationsleiste des Cores ein Icon hinzuzufügen sowie eine Seite, welche beim Klicken auf das Icon geladen werden soll.
Um jedoch den vollen Umfang der Funktionalität der Konferenz zu gewährleisten, muss es einem Modul möglich sein, auf Zustände, welche im Redux-Store der Hauptanwendung liegen, zuzugreifen. Die in Listing 47 angegebene Komponente benötigt diese Funktionalität. Im Core liegt, wie in Abschnitt 4.4 erwähnt, der Provider auf der obersten Ebene, sodass der Store alle Komponenten einschließt. Dies muss nun auch für die *CreateMeeting* Komponente des Moduls gelten. Um dies zu ermöglichen, wird ein Wrapper für die Komponente geschrieben, welcher die Komponente in einen Provider packt. Dieser Komponente wird nun ein Store übergeben, welcher dann dem Provider übergeben wird.

```
export const CreateMeetingWrapper = ({
  store,
}) => {
  return (
    <Provider store={store}>
      <CreateMeeting />
    </Provider>
  );
};
```

Listing 48: Implementierung eines Wrapper zu Nutzung des Stores

In dem Core werden die Seiten, zu welchen man über die Navigationselemente gelangt, aus einem Array mit Elementen des Typs Route generiert. Dort kann nun jeder Komponente das Store-Objekt des Cores übergeben werden. Auf diese Weise kann eine Komponente eines Moduls mittels der useSelector-Hook auf den Store des Cores zugreifen.

Zuletzt gibt es, wie in Abschnitt 4.2 erwähnt, bestimmte Komponenten, die nicht in die Library ausgelagert wurden. Diese sollen dennoch in Modulen verwendet werden können. Dazu kann die in Abschnitt 2.3 beschriebene Funktion *lazy* von React verwendet werden. Hierbei muss der import-Funktion der in der Konfiguration festgelegte Name, sowie der öffentliche Pfad angegeben werden.

```
const CustomPromt = React.lazy(() => import("core/components"));
```

Listing 49: Laden einer im Core liegenden Komponente

5 Fazit

Die in Abschnitt 1.3 formulierten Fragen können nach Abschluss dieser Arbeit eindeutig beantwortet werden. Demnach hat die Konzeptionierung gezeigt, dass sich Konzepte aus vorhandenen Plugin-Systemen, welche sich jedoch technisch von einer in dieser Arbeit beschriebenen Anwendung unterscheiden, übernehmen lassen. Dabei wurde sich bei der Umsetzung vor allem an dem Plugin-System von Visual Studio Code orientiert, welches technisch der Anwendung Audimax am nächsten liegt. Das Vorgeben einer Ordnerstruktur sowie das Importieren eines Moduls, um eine API für die Hauptanwendung bereitzustellen, wurden dabei übernommen.

Der Großteil der in Abschnitt 3.1 gestellten Anforderungen wurde von dem Konzept umgesetzt. Erweiterungen ist es möglich, der Hauptanwendung weitere Funktionalitäten hinzuzufügen. Dafür muss aufgrund der Core API nicht der Quellcode der Hauptanwendung verändert werden. Somit haben Erweiterungen auch keinen Einfluss auf die Funktionalität der Hauptanwendung. Lediglich die Möglichkeit, Erweiterungen ebenfalls zu erweitern, wurde in diesem Konzept nicht umgesetzt. Dies liegt jedoch nicht daran, dass dies technisch nicht möglich ist. Vielmehr fehlte die Zeit, dies noch im Rahmen dieser Arbeit umzusetzen. Eine Idee der Umsetzung der Anforderung wird in Abschnitt 5.1 kurz erläutert.

Durch Module Federation war es möglich, eine Technologie in das bestehende System einzuführen, ohne dass ein Umstieg auf ein weiteres Framework oder der Schritt zu einer auf dem Server gerenderten Anwendung notwendig war. Dies vereinfachte vor allem die Implementierung des konzeptionierten modularen Systems, da dadurch nicht die Grundstruktur der Anwendung verändert werden musste, sondern die gewünschten Module nacheinander aus der Hauptanwendung in ein eigenes Modul ausgelagert werden konnten.
Aufgrund der neuartigen Technologie kam es jedoch gerade zu Beginn der Entwicklung zu Schwierigkeiten, da das Tool Create React App die Unterstützung von Module Federation noch nicht veröffentlicht hat (*Stand: 29.11.21*). Letztendlich konnte jedoch mithilfe des *craco-module-federation*-Plugins Module Federation der Anwendung hinzugefügt werden.

Eine Veröffentlichung der Anwendung Audimax mit unterstützter Modularisierung würde nach der in dieser Arbeit beschriebenen Entwicklung jedoch noch warten müssen, da auch das craco-module-federation-Plugin noch nicht für den Produktiveinsatz empfohlen wird. Sollte eine zeitnahe Veröffentlichung dennoch vonnöten sein, so könnte auf Create React App verzichtet werden. Dies würde einen problemlosen Produktiveinsatz der modularisierten Anwendung ermöglichen.

Nach erfolgreicher Auslagerung der Konferenz in ein eigenes Modul kann jedoch die Prognose gegeben werden, dass es möglich ist, mithilfe von Module Federation eine Single-Page-

Application zu modularisieren und somit die gewünschten Vorteile, welche im Abschnitt 1.1 erläutert wurden, zu bieten.

5.1 Ausblick

Die in dieser Arbeit gezeigte Umsetzung der Modularisierung beschränkte sich auf das Modularisieren der Konferenz. Das Auslagern anderer Funktionalitäten kann dabei in Zukunft umfangreicher ausfallen, sodass die Entwicklung der Core API und somit ebenfalls des extensionObjects sich den dadurch gegeben Umständen anpassen muss. Zusätzlich könnte in Zukunft eine Funktion benötigt werden, welche vor dem Aufruf der activate-Methode ausgeführt wird, um zum Beispiel bestimmte Typen oder Funktionen zuerst zu registrieren, damit diese beim Ausführen der activate-Methode zur Verfügung stehen.
Ebenfalls wurde in dieser Arbeit noch nicht konzeptioniert, wie ein Modul ein weiteres Modul erweitern kann. Hierbei sollte in Zukunft die Möglichkeit für Module bestehen, Funktionen im Core zu registrieren, sodass diese durch andere Module aufgerufen werden können.
Zuletzt soll auch die Erstellung eines Moduls einem Entwickler erleichtert werden. So soll diesem ein CLI-Tool bereitstehen, welches die Grundstruktur sowie die grundlegenden Konfigurationen vornimmt und dadurch die Entwicklung erleichtert.

Literaturverzeichnis

[Abr19] Abramov, Dan: React v16.8: The One With Hooks (2019), URL https://reactjs.org/blog/2019/02/06/react-v16.8.0.html, Abrufdatum: 29.11.2021

[Aya] Ayan, Hasan: add module federation support, URL https://github.com/facebook/create-react-app/pull/11241, Abrufdatum: 26.11.2021

[Bue18] Buehler, Peter; Schlaich, Patrick und Sinner, Dominik: *Webtechnologien : JavaScript – PHP – Datenbank*, Bibliothek der Mediengestaltung, Springer Berlin Heidelberg, Berlin, Heidelberg, 1st ed. 2018 Aufl. (2018), URL https://doi.org/10.1007/978-3-662-54730-4

[fDuI21] für Datenschutz und Informationsfreiheit, Der Hessische Beauftragte: Kein ausreichender Datenschutz bei der Nutzung von Padlet (2021), URL https://datenschutz.hessen.de/datenschutz/hochschulen-schulen-und-archive/kein-ausreichender-datenschutz-bei-der-nutzung-von

[Gso] Gsoft: CRACO, URL https://github.com/gsoft-inc/craco/blob/master/README.md, Abrufdatum: 25.11.2021

[Har17] Harris, Rich: Webpack and Rollup: the same but different (2017), URL https://medium.com/webpack/webpack-and-rollup-the-same-but-different-a41ad427058c

[Har20] Hartmann, Nils und Zeigermann, Oliver: *React : Grundlagen, fortgeschrittene Techniken und Praxistipps - mit TypeScript und Redux*, Heidelberg, 2., überarbeitete und erweiterte auflage Aufl. (2020)

[Jaz07] Jazayeri, Mehdi: Some Trends in Web Application Development, in: *Future of Software Engineering (FOSE '07)*, S. 199–213

[JH20] Jack Herrington, Zack Jackson: *Practical Module Federation*, Jack Herrington, Zack Jackson (2020)

[Maj] Maj, Wojciech: React lifecycle methods diagram, URL https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/, Abrufdatum: 25.11.2021

[McV09] McVeigh, Andrew: A Rigorous, Architectural Approach to Extensible Applications (2009)

[Nex] NEXTCLOUD.COM: Introduction, URL https://docs.nextcloud.com/server/latest/developer_manual/app_development/intro.html, Abrufdatum: 29.11.2021

[Ove21] OVERFLOW, Stack: Stack Overflow Developer Survey 2021, https://insights.stackoverflow.com/survey/2021#most-popular-technologies-webframe (2021)

[Reaa] REACTJS.ORG: Code-Splitting - React, URL https://reactjs.org/docs/code-splitting.html#import, Abrufdatum: 20.11.2021

[Reab] REACTJS.ORG: Code-Splitting – React, URL https://reactjs.org/docs/code-splitting.html#import, Abrufdatum: 22.11.2021

[Reac] REACTJS.ORG: DOM Elements, URL https://reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml, Abrufdatum: 29.11.2021

[Read] REACTJS.ORG: Hooks API Reference – React, URL https://reactjs.org/docs/hooks-reference.html#cleaning-up-an-effect, Abrufdatum: 25.11.2021

[Reae] REACTROUTER.COM: React Router: Declarative Routing for React.js, URL https://v5.reactrouter.com/web/api/BrowserRouter, Abrufdatum: 29.11.2021

[Ste16] STEINER, Tobias: Das Wordpress-Kompendium (2016), URL http://nbn-resolving.de/urn:nbn:de:101:1-2016102113218

[Vis] VISUALSTUDIO.COM: Extension Anatomy, URL https://code.visualstudio.com/api/get-started/extension-anatomy#extension-file-structure, Abrufdatum: 29.11.2021

[Weba] WEBPACK.JS.ORG: Tree Shaking, URL https://webpack.js.org/guides/tree-shaking/, Abrufdatum: 29.11.2021

[webb] WEBPACK.JS.ORG: webpack, URL https://webpack.js.org/, Abrufdatum: 20.11.2021

[Web20] WEBPACK: Webpack 5 release (2020-10-10), https://webpack.js.org/blog/2020-10-10-webpack-5-release/#automatic-nodejs-polyfills-removed (2020)

[Wora] WORDPRESS.ORG: Header Requirements, URL https://developer.wordpress.org/plugins/plugin-basics/header-requirements/, Abrufdatum: 29.11.2021

[Worb] WORDPRESS.ORG: Introduction to Plugin Development, URL https://developer.wordpress.org/plugins/intro/, Abrufdatum: 29.11.2021