Dash.js Reference Client

Johannes Beyer
Elektrotechnik M.Sc.
Fakultät IV
Technische Universität Berlin
johannes.beyer@campus.tu-berlin.de

Berit Frech
Information System Management M.Sc.
Fakultät IV
Technische Universität Berlin
berit.frech@campus.tu-berlin.de

Anna Sophie Mockenhaupt
Medieninformatik M.Sc.
Fakultät IV
Technische Universität Berlin
a.mockenhaupt.1@campus.tu-berlin.de

***Abstract*—Die Arbeit befasst sich mit dem Entwurf und der Entwicklung einer neuen Referenz UI für den dash.js Client. Das Ziel des Projektes ist es, eine neue Referenz UI zu implementieren, welche die Konfiguration der dash.js Einstellungen ermöglicht.**

***Index Terms*—dash.js, angular, adaptive bitrate streaming, referenz UI, DASH-Player**

I. Einleitung

A. Grundlagen

Der MPEG-DASH-Player dash.js ist ein JavaScript Referenz Client, welcher zur Implemetierung von DASH-Playern genutzt wird. Er basiert auf den Media Source Extensions (MSE) und den verschlüsselten Medien Erweiterungen (EME). [1].

Das Hauptziel von dash.js ist der Aufbau einer Open-Source-JavaScript-Bibliothek für die Wiedergabe von Dynamic Adaptive Streaming über HTTP (MPEG-DASH). MPEG-DASH ist ein adaptiver Streaming-Standard für die Bereitstellung von Streams an Benutzer auf verschiedenen Plattformen. In MPEG-DASH werden verschieden Versionen einer Videodatei mit unterschiedlichen Auflösungen erstellt. Jede dieser Versionen wird in Segmente aufgeteilt. In einer Media Presentation Description (MPD) Datei, werden die Details dieser Versionen gespeichert.

sBei der Wiedergabe eines Streams, wird die MPD Datei heruntergeladen. Die Dateiinhalte werden analysiert, um die geeignete Version, welcher der Internetverbindung enstpricht, zu laden. Sollte sich die Internetverbindung während der Wiedergabe der Datei verändern, wechselt der Player automatisch auf die Version, welche dem neuen Status entspricht. So wird ein Anhalten der Wiedergabe verhindert. [2]

B. Probleme der alten UI

Die aktuelle Referenz UI basiert auf AngularJS, einem JavaScript Framework, welches nach drei Jahren Long Term Support (LTS) ab dem 31.Dezember 2021 nicht mehr unterstützt werden soll. Ebenfalls sind die eingebundenen Libraries veraltet. Auch die Einstellungen des Players werden unvollständig und unübersichtlich auf der UI abgebildet. Außerdem wirkt der Bildschirm durch das veraltete Design überladen und unübersichtlich. Nicht nur einzelne Einstellungen, auch die Funktionsbeschreibungen fehlen oder sind nicht vollständig aufgeführt.

Die Referenz UI ist veraltet und unvollständig.

C. Zielsetzung

Das Ziel des Projektes war es, eine neue Referenz UI zu implementieren, welche die Konfiguration der dash.js Einstellungen ermöglicht. Alle möglichen Einstellungen von dash.js sollen integriert und die dazugehörigen API-Aufrufe wieder gespiegelt werden. Die Benutzerfreundlichkeit der UI soll überarbeitet und übersichtlicher gestaltet werden. Außerdem soll das Framework Angular in Version 11 verwendet werden. Ebenfalls soll die Library zur Abbildung der Metrics ausgetauscht werden.

II. Implementierung

A. Projektaufbau

1) Libraries: Die Basis der App bildet die aktuelle Version 11 des TypeScript-Frameworks Angular [3]. Der Umstieg auf TypeScript ermöglicht eine deutliche Verbesserung des Codes hinsichtlich Konsistenz, Modularität und Wartbarkeit. Gleichzeitig gibt die strenge Typisierung weniger Raum für unvorhergesehene Laufzeitfehler.

Die Library RxJS ermöglicht die Verwendung von Observables [6]. Sie sind unumgänglich, um eine service-basierte Kommunikation zwischen Angular-Komponenten zu implementieren.

Zusätzlich wird die Angular-Erweiterung Angular Material [4] verwendet. Es handelt sich hier um eine Library für UI-Komponenten, die sich an den Guidelines von material.io [5] orientiert und diese für Angular-Projekte einfach realisierbar macht. Verwendung fanden hauptsächlich die Form-Elemente für Inputs, Checkboxen, Slide-Toggles und ein Slider zur Einstellung der Video-Qualität. Auch einige Cards, Expansion Panels, eine Menu-Komponente (Auswahl zur Gruppierung der Streams), eine Snackbar (Anzeige, dass max. 5 Metrics ausgewählt werden können) und ein Dialog (Einstellung von DRM-Daten) wurden implementiert.

Die Umsetzung des Layouts erfolgt auf Basis von Bootstraps Grid-System [7]. Dies ermöglicht eine einfache und flexible Anpassung der Darstellung für jede gängige Bildschirm-Größe bzw. -Auflösung. Das Design ist vollständig responsive und ermöglicht eine Nutzung auch auf mobilen Geräten. Verwendet werden auch vereinzelt Design-Elemente aus Bootstrap (z. B. Badges) bzw. weitere CSS-Klassen. Jedoch beschränken wir uns hier auf Design-Elemente.

Eine besondere Herausforderung stellt die kompakte Darstellung der großen Anzahl an Settings dar. Diese sind

in Gruppen unterteilt und können dynamisch ein- oder ausgeklappt werden. Die Library ngx-masonry [8] ordnet die Blöcke dynamisch in der jeweils passenden Art an.

Die Darstellung der Metrics erfolgt über eine Chart-Library. Hier gestaltete sich die Auswahl durchaus schwierig. Gesucht war eine Library, die eine Möglichkeit zur Darstellung von Live-Daten und Datenreihen mit mehreren Y-Achsen bietet. Damit fielen die drei bekanntesten canvasJS, ng2-charts, ngx-charts bereits weg. Die Lib Flot (alte UI) schied zunächst aus, da sie vermeintlich stark veraltet war. Wie sich später herausstellte, ist im Github Repository lediglich das Tag "Latest Release" falsch gesetzt [11]. Lediglich Plotly und Apexcharts bieten alle notwendigen Features und stellten sich als aktuell dar. Plotly basiert auf Javascript und stellt sich sehr komplex dar, während Apexcharts auf Basis von Typescript eine bestmögliche Einbindung in Angular ermöglicht und, gemessen am Funktionsumfang, vergleichsweise einfach in der Anwendung erscheint. Die Wahl fällt klar auf Apexcharts [9]. Allerdings unterstützt diese Lib offiziell noch nicht Angular in der neuesten Version 11. Das Paket muss dementsprechend mit dem Schalter –force installiert werden. In der weiteren Entwicklung der App zeigten sich keine Probleme mit Apexcharts unter Angular v. 11.

2) Komponenten: Der Aufbau der Angular-App folgt einem gewissen Baukasten-Prinzip. Die Datei src/index.html stellt dabei das Basis-Template. Hier wird der grundlegende HTML-Baum definiert. Über das Tag *<app-root></app-root>* wird die eigentliche App strukturell eingebunden. In der Datei src/styles.css werden globale CSS-Regeln definiert. Die Elemente des Baukastens sind stets App-Komponenten. Eine Komponente wird in einer TypeScript-Datei definiert und setzt sich aus Imports, Decorator und Klassen-Definition zusammen. Über den Decorator wird der Selector definiert, über den die Komponente in einem HTML-Template eingebunden wird. Bspw. verwendet die Root-Komponente den Selector app-root und wird daher mit *<app-root></app-root>* in der src/index.html eingebunden.

Listing 1. Angular Komponente

```
import {...} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  ...
})
export class AppComponent {
  ...
}
```

Außerdem werden im Decorator Templates und Styles der Komponente definiert. So wird für die Root-Komponente die Datei src/app/app.component.html als Template festgelegt. Über das Template können neben normalen HTML-Anweisungen andere Komponenten über ihren jeweiligen Selector eingebunden werden.

Listing 2. Template der Root-Komponente

```
<app-header></app-header>
<app-video-configuration></...>
<div>
  <div class="row">
    <div class="...">
      <app-player></app-player>
    </div>
    <div class="...">
      <app-metrics-configuration></...>
    </div>
  </div>
</div>
<app-metrics-view></app-metrics-view>
<app-footer></app-footer>
```

Der Reference-Client ist entsprechend seiner funktionellen Struktur in Komponenten aufgeteilt, welche sich unter src/app/components/ befinden und von der Root-Komponente eingebunden werden. Neben den trivialen Komponenten für Header und Footer gibt es eine Komponente "player", die über das HTML5 <video>-Tag den dash.js-Player in die App einbettet. Die Komponente "video-configuration" stellt einen Bereich zur Eingabe / Auswahl eines Streams zur Verfügung und bindet des Weiteren die Tochter-Komponente "setting" ein, welche alle verfügbaren Settings gegliedert darstellt und Änderungen verarbeitet. Die Komponente "metrics-configuration" listet verfügbare Stream-Metrics auf und stellt ähnlich zur vorherigen UI Live-Werte der Metrics kompakt dar. Außerdem können hier Metrics zur Anzeige in einem Live-Chart ausgewählt werden. Das Live-Chart wird in der Komponente "metrics-view" eingebunden.

3) Services: Die Einbindung des dash.js-Players erfolgt über den Service PlayerService (src/app/services/player.service.ts). Dieser erzeugt über *dashjs.MediaPlayer().create()* das Player-Objekt, speichert es und übergibt es ggf. den App-Komponenten. Außerdem werden hier Helfer-Methoden zur Steuerung des Players oder Berechnung der Metrics zur Verfügung gestellt. Die Initialisierung bspw. erfolgt in der Komponente 'player' über den Befehl *this.playerService.initialize(this.videoElement.nativeElement);*. Der PlayerService steuert auch die Akamai-Toolbar. Diese liegt nicht als Typescript-Modul vor. Daher wird die Datei ControlBar.js über die Konfiguration angular.json direkt eingebunden.

Der Service MetricsService (src/app/services/metrics.service.ts) ermöglicht die Kommunikation zwischen den Komponenten "metrics-configuration" und "metrics-view" (siehe auch II-C). Zur Kommunikation zwischen Angular-Komponenten kommt bei Komponenten mit einer Parent-Child-Beziehung typischer Weise "Input Property Binding" zum Einsatz. In einer Sibling-Anordnung ist dies jedoch nicht möglich. Hier wird ein Service benötigt (siehe auch [10]).

4) Types: Wie sich im Verlauf der App-Entwicklung zeigte, sind einige Type-Deklarationen des Moduls dash.js unvollständig. Konkret fehlt in der index.d.ts unter *export interface DashAdapter*{...} die Deklaration der Methode *getPeriodById(...)*, welche zur Berechnung einiger Metrics benötigt wird. Abhilfe schafft hier eine Modul-Augmentierung. Dabei werden in der Datei src/app/types/dashjs-types.ts mittels der Anweisung *declare module 'dashjs'* {...} eigene Deklarationen zum vorhandenen Modul dashjs ergänzt. Neben der genannten Funktion wurden hier auch die Interfaces *Period* und *HTTPRequest* definiert.

In der Datei src/app/types/metric-types.ts wurden außerdem einige wiederkehrende komplexe Interfaces für Metric-Objekte definiert.

5) Assets: Unter src/assets/ finden sich des Weiteren Definitionen von Konstanten für Metrics, Setting-Gruppierung und -Beschreibungen, sowie einer Helfer-Funktion *hasownproperty()*. Diese erweitert die Funktionalität der gleichnamigen Javascript-Funktion um eine Typ-Eingrenzung, um nach Prüfung auf eine Objekt-Property auch typsicher darauf zugreifen zu können. Ebenfalls befinden sich im Assets-Ordner die Akamai-Toolbar, verwendete Grafiken und die JSON-Dateien contributors.json und sources.json, welche Daten zu in der App angezeigten mitwirkenden Unternehmen bzw. auswählbaren Beispiel-Streams und deren Providern bereitstellen. Für einige Streams sind in der sources.json weitere Informationen wie DRM-Daten oder spezielle Buffer-Settings hinterlegt. Diese werden ebenfalls in den Komponenten "video-configuration" und "setting" übernommen und angezeigt.

B. Settings

Der Aufbau der Setting Optionen in der neuen UI erfolgt in zwei Phasen: zuerst werden die default Settings des dash.js Player Objects eingelesen, verarbeitet und formatiert. Anhand des formatierten Settings Objektes wird daraufhin das User Interface iterativ aufgebaut, wobei die drei möglichen Input Elemente (Radio-Buttons, Input Fields und Checkboxen) dynamisch anhand des Typs bestimmt werden.

1) Verarbeitung der default Setting: Das Einlesen und Verarbeiten der default Settings ist in der video-configuration Komponente implementiert.

Zunächst wird das verschachtelte Objekt der default Settings mit der *flattenObject* Funktion iteriert und in ein Array gespeichert. Dieses Array beinhaltet neben der Gruppenzuordnung, dem Namen und dem Wert der Einstellung auch den Pfad der einzelnen Einstellungen durch das verschachtelte Objekt.

Die Gruppenzuordnung wird im folgenden Schritt genutzt um zusammenhängende Settings zu kombinieren. Als zusammenhängend werden alle Optionen angesehen, die mehr als drei Verschachtelungen aufweisen. So werden beispielsweise Settings, die für Audio und Video separat gesteuert werden können, in der UI als eine Gruppe dargestellt.

Daraufhin folgt ein mapping der default Settings auf die individuell eingestellte Gruppierung, die anhand des *settingGroups* Objektes festgelegt ist (die settingGroups.ts Datei befindet sich im 'asset' Ordner der Projektes). Werte, die nicht in der UI angezeigt werden sollen, können in die 'NONE' Gruppe zugeordnet werden. Optionen, die sich in den default Settings des Players befinden, aber noch nicht manuell einer Gruppe in den settingGroups zugeordnet wurden, werden automatisch der Gruppe 'UNASSIGNED' zugeordnet.

Zuletzt wird das Objekt anhand der individuellen Zuordnung gruppiert und zurückgegeben.

Listing 3. Logischer Aufbau des Setting-Objektes nach Verarbeitung

```
groups:
    DEBUG:
        DispatchEvent:
            value: false,
            path: debug.dispatchEvent,
        LogLevel:
            value: 3
            path: debug.LogLevel
        MetricsMaxListDepth:
            value: 1000,
            path: streaming.metricsMaxListDepth
    ABR:
        ABRStrategy:
            value: abrDynamic,
            path: streaming.abr.ABRStrategy,
        ...
    ...
```

2) Aufbau des Templates: Das Objekt mit der finalen Gruppierung wird daraufhin als Array (basierend auf groups) in der setting Komponente eingelesen und iteriert. Für jede Gruppe wird ein Masonry Item erstellt, in welches die dazugehörigen Settings dynamisch geladen werden. Die dynamischen Settings werden ergänzt durch fest codierte Einstellungsmöglichkeiten zur DRM Verschlüsselung, Text Einstellungen, Qualität, sowie Loop und Autoplay.

Beim Einlesen der dynamischen Settings unterscheiden wir zwischen drei Typen: Boolean, Number und Konstanten, die dementsprechend als Checkboxen, Input Felder oder Radio Buttons aufgebaut werden. Der Typ wird anhand des default Wertes der Einstellung geprüft. Input Felder und Checkboxen werden allein mit dem Namen und dem default Wert des *group* Arrays aufgebaut. Um die möglichen Werte der Konstanten in Radio Buttons zu übersetzen, wird zusätzlich das *constants* Objekt genutzt. Dies beinhaltet die möglichen Werte, sowie deren default Wert (true or false) und muss manuell eingepflegt werden.

Zusammenhängende Settings werden nach dem gleichen Prinzip mit einer weiteren Iteration aufgebaut.

Falls diese vorhanden sind, werden die API Beschreibungen der einzelnen Optionen über das *settingGroups* Objekt dynamisch ausgelesen und in die UI eingefügt.

Die 'PLAYBACK' Gruppe wird um die fest codierten Einstellungen zur Qualität, Loop und Auto Playback erweitert, und die 'INITIAL' Gruppe um die Text Einstellungen, da diese nicht Teil der default Settings sind und daher nicht dynamisch geladen werden.

Die DRM Einstellungsmöglichkeiten wurden durch ein Eingabefeld erweitert, welches JSON Objekte akzeptiert. Dies ermöglicht das Anhängen einen Protection Data Sets an einen

Stream, sowie das Auslesen bereits existierender Protection Data Sets für den Fall, dass der geladene Stream bereits Daten zur DRM Verschlüsselung beinhalten.

3) Update Funktion: Änderungen der Settings über die UI werden durch einen Aufruf der update Funktion in der setting Komponente an das Player Objekt übergeben. Der Aufruf erfolgt mit dem Pfad und dem Eingabewert als Input, wobei der Pfad aus dem *group* Array ausgelesen wird. Anhand des Pfades wird ein verschachteltes Objekt mit dem neuen Wert aufgebaut, welches dann an die *updateSettings* Funktion des dash.js Player übergeben wird. Dies ermöglicht es, die selbe Funktion für alle dynamischen Settings der UI zu verwenden.

Die Checkbox "use Default ABR Rules" ruft zusätzlich zu *update* die Funktion *toggleABRRules* auf, die entsprechend des Wertes der Checkbox die manuell erstellten Regeln hinzufügt oder entfernt. Durch den Umstieg auf Typescript im Rahmen unserer Implementierung, sollten diese Regeln in Typescript formuliert und eingefügt werden. Alternativ kann eine type-declaration hinzugefügt werden.

Ein Spezialfall ist die LogLevel Einstellung der DEBUG Gruppe, da hier nicht Strings sondern die dazugehörigen numerischen Werte des Enums an die *updateSettings* Funktion übergeben werden. Einstellungsänderungen des Log Levels werden daher an die *updateLogLevel* Funktion übergeben.

C. Metrics

Metrics eines geladenen Streams sollen analog zum alten Reference Client in einem Live-Chart dargestellt werden. In einem Bereich der App werden verfügbare Metrics aufgelistet und über Checkboxen auswählbar gemacht. Außerdem werden hier die aktuellen Werte kompakt angezeigt. Dieser Bereich wurde als Komponente "metrics-configuration" implementiert. In einem anderen Bereich wird ein Live-Chart entsprechend der ausgewählten Metrics angezeigt. Die Komponente heißt "metrics-view". Von der Funktionslogik her bietet sich hier eine Parent-Child-Beziehung der Komponenten an. Diese würde auch die einfache Kommunikation zwischen den Komponenten ermöglichen. Durch die Trennung im Template ist das jedoch nicht möglich. Beide Komponenten können nur in einer Sibling-Konfiguration angeordnet werden. Für die Kommunikation muss daher ein Service "metrics.service" implementiert werden. Die Berechnung der Messwerte erfolgt im Service "player.service". Diese wurden hauptsächlich aus dem alten Reference Client übernommen. Der Code wurde dazu typsicher in TypeScript in den Funktionen *PlayerService.getMetrics(...)* und *PlayerService.calculateHTTPMetrics(...)* refaktorisiert. Die HTTP-Metrics werden allerdings nun in Millisekunden angegeben.

Im Folgenden soll der logische Programmablauf näher beschrieben werden. Die Komponente "metrics-configuration" importiert die Konstante *METRICOPTIONS* aus der src/assets/metrics.ts. Darin sind alle Messwerte aufgelistet, die von der UI berechnet und dargestellt werden können. Zu jedem Messwert ist hier ein Anzeigename, ein Schlüssel für den Datenzugriff und ein Typ hinterlegt. Für jeden hinterlegten Messwert erzeugt die Komponente nun eine Tabellenzeile mit dem Namen des Messwertes. Das Typ-Feld speichert die Information, auf welchen Medien-Typ sich die Metric bezieht. Mögliche Werte sind 'a', 'v', 'av' oder 'stream'. Bei 'a' (nur audio) erzeugt die Komponente eine Checkbox unter 'audio', bei 'v' (nur video)unter 'video' und bei 'av' (audio und video) unter beiden Medien-Typen. Die Angabe 'stream' führt zu einer mittig (zwischen 'audio' und 'video') angeordneten Checkbox. Wird eine Checkbox an- oder abgewählt, so führt das zum Aufruf der Methode optionChange(...). Diese stellt bei einer Anwahl sicher, dass maximal 5 Optionen aktiv sind und aktualisiert ggf. das Array selectedOptionKeys, welches den Key jedes ausgewählten Messwertes speichert. Sobald ein neuer Wert in das Array geschrieben oder ein alter Wert gelöscht wurde, wird außerdem die Service-Methode *this.metricsService.updateMetricsSelection(...)* aufgerufen und das Array übergeben.

Der Service "metrics.service" nutzt rxjs-Subjects bzw. Observables um Daten bzw. Methodenaufrufe zwischen Komponenten weiterzugeben. Dazu wird für jede weiterzugebende Methode ein privates Subject erzeugt und von jedem Subject ein Observable abgeleitet. Die aufgerufene Methode *updateMetricsSelection()* kann über einen Aufruf von *next()* des Subject-Objekts einen zu verteilenden Wert übergeben, der nun auch im Observable, hier *updateMetricsSelectionCalled$*, zur Verfügung steht. Die Komponente "metrics-view" importiert den Service und kann nun im Konstruktor über *this.metricsService.updateMetricsSelectionCalled$.subscribe(...)* einen Observer erzeugen, der für jedes im Service ausgeführte *next()* den entsprechenden Wert übernehmen oder Methoden der aktuellen Komponente aufrufen kann. In diesem Fall wird das Array in *this.selectedOptionKeys* abgelegt und die Methode *this.metricsSelectionChanged()* aufgerufen.

Des Weiteren bindet die Komponente "metrics-view" die Chart-Library ein und erzeugt ein Intervall, in dem alle 1000 ms die Methode *intervalMain()* aufgerufen wird. Diese prüft zunächst, ob dash.js initialisiert wurde und ruft ggf. die Methode *updateChartData()* auf. Diese ruft Messwerte von "player.service" ab und speichert sie in einem lokalen Objekt *chartData*. Wurden Messwerte zur Anzeige ausgewählt, so ruft die *intervalMain()* außerdem die Methode *updateChart()* auf. Sie erzeugt ein Messreihen- und ein Y-Achsen-Objekt und übergibt beide zur Aktualisierung an die Chart-Library. Die Chart-Aktualiserung wird nicht durchgeführt, wenn sich der Mauszeiger im Chart-Container befindet. In diesem Fall wird pausiert und ermöglicht dem Benutzer so einige Interaktionen um z. B. genaue Werte zu erhalten oder eine Messreihe hervorzuheben.

D. Benutzerfreundlichkeit

1) Benutzeroberfläche: Die Benutzeroberfläche gliedert sich in vier Bereiche. Anhand der Übersicht in Fig 1 (*Übersicht über die Benutzeroberfläche der Referenz UI)*. werden die wichtigsten Anpassungen erläutert.

Sortierung der Streams: Den ersten Bereich **(1)** bilden die Filebar und das Dropdown-Menü der Streams ab. Über

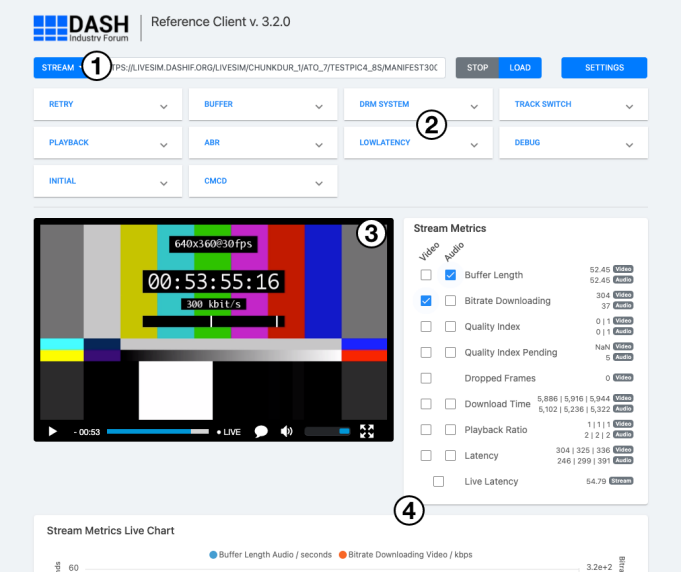

Fig. 1. Übersicht über die Benutzeroberfläche der Referenz UI

das Dropdown-Menü oben links können die zur Verfügung stehenden Streams angezeigt und in die Filebar eingeladen werden.

Initial sind die Streams nach Stream-Typ sortiert und in Akkordeons gruppiert. Neben dem Namen des Streams, ist außerdem der Namen des Providers aufgelistet. Zur optisch besseren Unterscheidbarkeit, werden die verschiedenen Provider mit farblich voneinander abgesetzten Badges markiert. Die Badges besitzen jeweils einen Tooltip mit der vollständigen Information zu dem Provider. Außerdem wird der Benutzerin bzw. dem -nutzer die Möglichkeit geboten über ein Submenü die Streams auch nach Provider sortiert anzuzeigen.

Der ausgewählte und aktive Stream wird, für die Benutzerin bzw. den -nutzer ersichtlich farblich unterlegt.

Im Projekt wird diese Funktion von der Komponente VideoConfigurationComponent bereitgestellt.

Gruppierung der Settings: Der zweite Teil umfasst die Settings **(2)**. Über den Menü Button oben rechts öffnen sich unter der Filebar und dem Settings Button die definierten Settings Gruppen (siehe Abschnitt B Settings). Jede Gruppe wird initial in einem geschlossenen Expansion Panel angezeigt, sodass die Benutzerin bzw. der -nutzer zu Beginnn eine übersichtliche Ansicht, der zur Vefügung stehenden Einstellungen bekommt.

Mit Klick auf eine Setting-Gruppe öffnet sich das Expansion Panel und die Benutzerin bzw. der -nutzer kann die zur Gruppe sortierten Einstellungen auswählen und bedienen. Die Expansion Panels haben eine fest definierte maximal Höhe, sodass die Ansicht nicht zu unruhig wirkt. Sollte der Inhalt der Gruppe über die maximale Höhe hinausgehen, wird eine Scrollbar innerhalb des Expansion Panels eingeblendet. Die Expansion Panels sind mit der grid layout library Masonry aufgebaut. Die Expansion Panels können so ansprechend und platzsparend in der Referenz UI angezeigt werden und es entsteht kein überflüssiger Weißraum.

Die Benutzerin bzw. der -nutzer hat die Möglichkeit mehrere Expansion Panels zur selben Zeit zu öffnen. Über das Ein und Ausklappen der gewünschten Settings ist es nun möglich nur die Einstellungen zu öffnen die für das gewählte Projekt relevant sind. Die Ansicht der Settings ist damit sehr übersichtlich und flexibel anpassbar.

Videoplayer: Im Zentrum der Referenz UI steht der Video Player **(3)**. Der Player besitzt eine Controllbar mit den Funktionen das Video zu starten und zu pausieren, einem Caption-button, Volume Regler inklusive Mute-Funktion und der Möglichkeit das Video auf Fullscreen anzuschauen.

Metrics: Den vierten Teil bilden die Metrics ab **(4)**. Auf der rechten Seite neben dem Player sind die möglichen Stream Metrics für Audio und Video nebeneinander aufgelistet. Über Checkboxen können maximal 5 Metrics ausgewählt werden, welche dann im unteren Teil der Referenz UI in einem Diagramm dargestellt werden. Die Komponente MetricsViewComponent bildet dieses Diagramm ab und kommuniziert mit der Komponente MetricsConfigurationComponent.

2) Angular Material: Die UI Komponenten Bibliothek, Angular Material bietet eingebaute Module fur das eigene Projekt. Um Angular Materials zu nutzen, muss das @angular/materials Paket importiert werden. Die, bereits vorgestylten und animierten, Materials konnen einfach mit den entsprechenden HTML Tags in das Template eingebunden werden. Um die Material Komponenten zu nutzen wird in der app.module.ts Datei noch das dazugehörige Modul importiert. Um die Materials zu stylen, kann die entsprechende Klasse in der globalen style.css angepasst werden.

Sollen sich die Attribute nur für eine bestimmte Komponente anpassen, kann die Verkapselung nur für diese Komponente deaktiviert werden. Dies geschieht mit im Decorator @Component über die property encapsulation: ViewEncapsulation.None. Im zugehörigen der zugehörigen Css Datei, wird das entsprechende Element gestylt. Die Klassen erlauben es die Materials nur für diese Komponente zu verändern und lassen den globalen Stil des Elements für das Projekt unverändert.

3) Styling: Jede Komponente verfügt über eine Css Datei. Innerhalb dieser werden die entsprechenden HTML-Elemente angepasst. Trotz der teilweise deaktivierten Verkapselung, soll der Grundgedanke von Angular beibehalten werden. Jedes Template wird mit einer eindeutigen id versehen, welche sich im präfix mit der Komponente in Verbindung stellen lässt. So wird in der CSS Datei jedes Element zuerst mit der id des Templates und dann mit der entsprechend gesetzten Klasse angesprochen werden.

In der globalen CSS-Datei werden weiterhin die root-Farben, container Größen und Überschriften gesammelt definiert. Diese Einstellungen gelten für das gesamte Projekt und sollen nicht pro Komponente variieren.

III. Evaluation

Unsere Implementierung des Reference Players bietet übersichtliche und vollständige Einstellungsmöglichkeiten für die Wiedergabe von MPEG-DASH Inhalten durch Gruppen, die ein- und ausgeblendet werden können. Die Anzahl und Benennung der Gruppen, sowie die Zugehörigkeit einzelner Einstellungen, kann unkompliziert über die Konstante *settingGroups* gesteuert werden, ohne dass Änderungen in den Komponenten vorgenommen werden müssen.

Außerdem bietet die *settingGroups* Konstante eine einfache Möglichkeit Beschreibungen oder Erklärungen zu einzelnen Einstellungen hinzuzufügen, die bei mouseover über das Fragezeichen Symbol angezeigt werden, und dadurch die UI verständlicher machen.

Durch das dynamische Aufbauen des Templates anhand der default Settings des Player Objektes werden alle Einstellungen, die im Settings.js file des dash.js Players implementiert sind, in der UI angezeigt. Sollten neue Einstellungen hinzugefügt werden, werden diese automatisch der Gruppe 'UNASSIGNED' hinzugefügt, bis eine manuelle Zuordnung in der *settingGroups* Konstante erfolgt.

Das Einfügen neuer Radio Buttons benötigt in unserer Implementation eine zusätzliche Definition der möglichen Werte und deren default Einstellung in der *constants* Konstante (das constants.ts file befindet sich im assets Ordner). Hier ist das Auslesen über die default Einstellungen nicht möglich, da dies nur Zugriff auf den aktuellen Wert erlaubt. Alternativ zur manuellen Definition in *constants* kann die Constant.js datei des dash.js Moduls (dashjs/src/streaming/constants/Constants.js) geparst werden. Dabei können alle Werte, die den Namen der einzelnen Einstellung beinhalten, gesammelt und in eine Konstante gespeichert werden. Dies würde einen dynamischen Aufbau der Radio Buttons ermöglichen, ist aber sehr Fehleranfällig, falls die Benennung sich leicht unterscheidet. Die Konstanten zur Einstellung 'movingAverageMethod' heißen beispielsweise 'MOVING_AVERAGE_SLIDING_WINDOW' und 'MOVING_AVERAGE_EWMA'. Eine suche nach 'moving Average Method' würde daher kein Ergebnis liefern. Daher haben wir uns in unserer Implementierung für die manuelle Eingabe über *constants* entschieden.

IV. Fazit

Im Rahmen dieses Projektes haben wir eine neue Reference UI für die Wiedergabe und Konfiguration von dash.js Inhalten implementiert. Diese basiert auf neuen Frameworks und ist durch dynamisches Einlesen der Settings vollständig und leicht zu erweitern.

Verglichen mit der alten UI bietet unsere Implementierung bessere Übersicht bei der Auswahl der Streams und der Einstellungen (im Vergleich mit Version 3.2.1 des Reference Client). Zusätzlich bieten wir eine einfache Möglichkeit Erklärungen zu den einzelnen Einstellungen hinzuzufügen und die Anordnung auf der UI anzupassen.

Ein vollständig dynamisches Laden der Radio Buttons ohne Nutzen von *constants* ist eine mögliche Erweiterung des Projektes. Zusätzlich würde ein einfacher Mechanismus zum Einfügen von "custom ABR Rules" über den Assets Ordner die Nutzerfreundlichkeit erhöhen.

References

[1] *Dash.js*, Fraunhofer FOKUS, März 2021, [online] Available: https://www.fokus.fraunhofer.de/en/fame/student/projects#Content-a7a97fe1

[2] *MPEG-DASH*, ArvanCloud, März 2021, [online] Available: https://www.arvancloud.com/de/products/video-platform/mpeg-dash

[3] *Angular.io*, Google, März 2021, [online] Available: https://angular.io

[4] *Angular Material*, Google, März 2021, [online] Available: https://material.angular.io

[5] *material.io*, Google, März 2021, [online] Available: https://material.io

[6] *RxJS*, März 2021, [online] Available: https://rxjs-dev.firebaseapp.com

[7] *Bootstrap*, März 2021, [online] Available: https://getbootstrap.com

[8] *ngx-masonry*, Wynfred, März 2021, [online] Available: https://www.npmjs.com/package/ngx-masonry

[9] *Apexcharts*, März 2021, [online] Available: https://apexcharts.com/docs/angular-charts/

[10] *Angular Component Interaction*, März 2021, [online] Available: https://angular.io/guide/component-interaction

[11] *Flot - Issue Latest Release*, NogginBox, März 2021, [online] Available: https://github.com/flot/flot/issues/1770