Progressive Web app for Media Streaming

Tarun Bhatia
Msc, Informatiks, TU Berlin
tarun.bhatia@campus.tu-berlin.de

Amit Nautiyal
Msc, Informatiks, TU Berlin
amit.nautiyal@campus.tu-berlin.de

Abstract—**Innovation in technology always makes an impact on how products and services are designed. The invention of Smartphone especially and after the launch of Android OS which is free there is a huge increase in the use of smartphones. Most of the users use native mobile applications for browsing the contents of the particular industry. The other way is to browse the contents is through a web browser. In order to overcome the limitations of web and native apps, Google has provided a solution of Progressive Web App (PWA) which combines the best of web and mobile apps giving us a rich experience just like the native apps. It is a website built using web technologies that acts like an app. PWA is a website built using web technologies that acts like an app and is not required to be installed like a native app. This document is intended to define the components and working of our project Progressive Web app for Media Streaming which intends to apply adaptive streaming of media and is mentored by Fraunhofer Fokus.**

Keywords : ,Progressive web application offline, service worker, app shell, app manifest

1. Introduction

In traditional mobile application development, reusability of code between native apps, the web, and mobile platforms has been inherently non-existent. This is due to native apps' non-interoperable code bases, resulting in separate projects and developer environments when support for multiple mobile platforms and operating systems is desired. For companies without the resources or the willingness to employ specialized mobile developers for each targeted platform, cross-platform development has become a popular alternative. a plethora of technical frameworks to support such development are either freely available under open source licenses, or as paid proprietary products. Ionic Framework and PhoneGap both belong to the web- and Cordova-based hybrid approach, where user interface components are solely structured and styled using web technologies including HTML and CSS.

Until recently, the web platform lagged behind on mobile-centred innovation in terms of being capable of competing with native or cross-platform apps (Puder et al., 2014). A new set of standards advocated by the Google Web Fundamentals group seeks to bridge that gap by introducing features such as offline support, background synchronisation, and home-screen installation to the web. The approach is known as Progressive Web Apps (PWA), a term coined by Russel and Berriman (2015) in a blog post covering initial design ideas. PWAs are defined by a set of concepts and keywords including progressive, responsive, connectivity independent, app-like, fresh, safe, discoverable, reengage able, installable, and linkable. These are PWA's contributions to the unification of the mobile experience, where web apps can be installed and distributed without app marketplaces, work without Internet connectivity, receive push notifications and look like regular apps. This is possible due to the Service Worker API which empowers developers through the use of a background-executed script acting as a network and device proxy.

2. Design Implementation

To gain better understanding of the possibilities of progressive web apps, three technical artefacts were developed for comparison. An hybrid app was developed using the Ionic Framework. An interpreted app was developed using React Native. Lastly, a progressive Web App was developed using React.js. The flow of Progressive web app can be understood by fig1.

2.1 Components of PWA

A typical Progressive web app has following mentioned components as depicted in the fig1.

2.1.1 Service Workers

A service worker is a type of web worker. It's essentially a JavaScript file that runs separately from the main browser thread, intercepting network requests, caching or retrieving resources from the cache, and delivering push messages. They are an incredibly powerful tool behind Progressive Web App.Service workers perform the following functionalities:

1. Caches the App Shell.
2. Updates the Content in background.
3. Gets the push notification id from the user to send the notification.
4. Invalidates the cache when needed

2.1.2 APP SHELL:

Application Shell Architecture is served up by the Service Worker and then the content is delivered. These are often cached by the service worker from its source through API requests. The sites that people visit more often will be able to

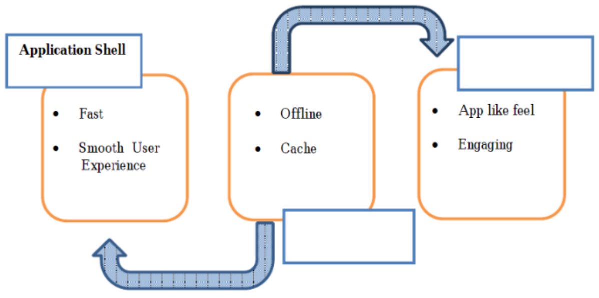


FIG 1: CORE TENETS OF PWA

hold the last content the person visited while waiting for the network to dynamically load the latest refresh.

2.1.3 WEB APP MANIFEST:

The web app manifest has the role to provide information about an application (such as its name, author, icon, and description) in a JSON text file. The manifest must inform details for websites installed on the home screen of a device, providing users with quicker access and a richer experience. The collection of web technologies called progressive web apps includes Web app manifests as its part through which websites that can be installed to a device's home screen without an app store, along with other capabilities like working offline and receiving push notifications.

3. Testing

Google Lighthouse can be run as a Chrome Extension, from the command line, or used programmatically as a Node module. Lighthouse focuses on performance metrics and the quality of PWAs. The standard Lighthouse report provides information on a wide array of factors, in an easily digestible format .All factors need to be fulfilled of a typical web application in order to qualify it as a Progressive Web Application. Some notable inclusions for website performance are:

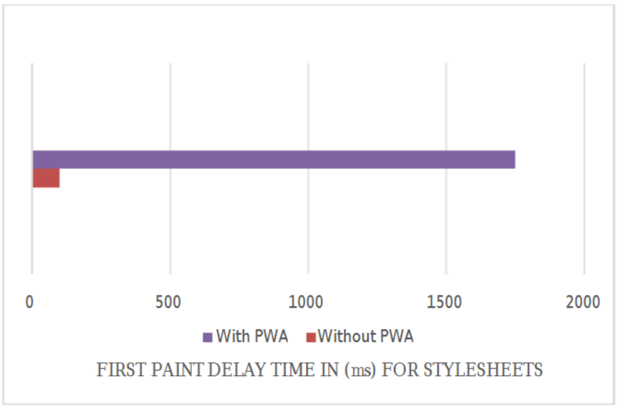


FIG 2: FIRST PAINT DELAY TIME

I. HTTP/2 – Provides a list of every internal resource that was not served over HTTP/2.

II. First Meaningful Paint – The time required to begin rendering content within the browser.

III. Time to Interactive – A measure of the point at which a page has loaded enough for a user to interact with it

First page is the first point where the browser has all the information that it needs to render the page. This is with

reference to the first time the browser will paint an image of the rendered page on the screen. Style sheets are a type of template file consisting of font and layout settings to give a standardized look to documents. The time taken for the first paint from the style sheet of PWA enabled webpage is smaller (85 ms) when compared to webpage without PWA (1769 ms).

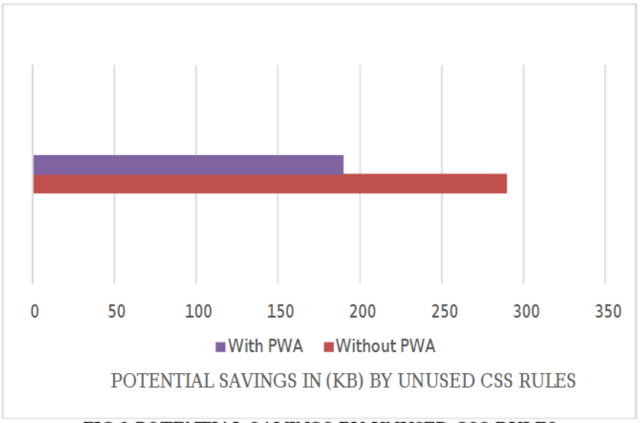


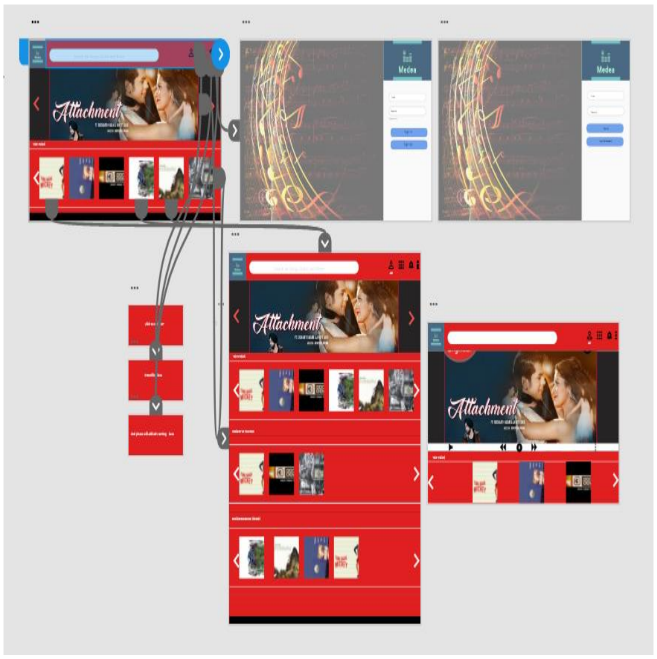
FIG 3:POTENTIAL SAVINGS BY UNUSED CSS RULES

4. Medea

Medea is a Progressive Web Application build for media streaming. The application incorporates all the typical functionalities of a PWA and uses adaptive media streaming to play videos in accordance with the network bandwidth available. It involves offline playback functionality with download capability for all media. Media player used is Shaka.

4.1 Wireframing

The wireframing done after understanding the requirements to plan the UI/UX design. Some of the wireframing components are shown in Fig 4 .Wireframing components are included in the final deliverables of the project.



4.2 Modern media streaming methods

From the early to late 2000s, video playback on the web mostly relied on the flash plugin. This was because, at the time, there was no other mean to stream video on a browser. As a user, you had the choice between either installing third-party plugins like flash or Silverlight, or not being able to play any video at all. Thus HTML5 brought, among other things, the <video> tag to the web. This new tag allows you to link to a video directly from the HTML, much like a <img> tag would do for an image. This HTML will allow your page to stream any video directly on any browser that supports the corresponding codecs.. This video tag also provides various APIs to e.g. play, pause, seek or change the speed at which the video plays. Those APIs are directly accessible through JavaScript:

```
//pause the video
myVideo.pause()

// seek to 10 seconds
myVideo.currentTime = 10;
```

Fig 5

However, most videos we see on the web today display much more complex behaviors than what this could allow. For example, switching between video qualities and live streaming would be unnecessarily difficult there.

Fig 6

YouTube displays some more complex use-cases: quality switches subtitles a tightly controlled progressive-download of the video. All those websites still use the video tag. But instead of simply setting a video file in the src attribute, they make use of much more powerful web APIs, the Media Source Extensions.

4.2.1 The Media source extension

The "Media Source Extensions" (more often shortened to just "MSE") is a specification from the W3C that most browsers implement today. It was created to allow those complex media use cases directly with HTML and JavaScript. Those "extensions" add the Media Source object to JavaScript. As its name suggests, this will be the source of the video, or put more simply, this is the object representing our video's data. Fig 7 shows the video is published to Media source which provides it to web page.

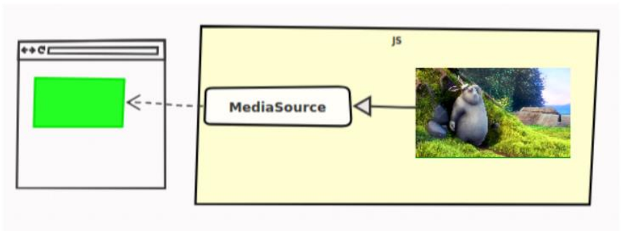


Fig 7

To allow this kind of use cases the W3C defined the URL. Create ObjectURL static method. This API allows creating an URL, which will actually refer not to a resource available online, but directly to a JavaScript object created on the client.

4.2.2 The Source Buffers

The video is not "pushed" into the Media Source for playback, Source Buffers are used for that. A Media Source contains one or multiple instances of those. Each being associated with a type of content .Source Buffers are all linked to a single Media Source and each will be used to add our video's data to the HTML5 video tag directly in JavaScript. A frequent use case (fig. 8) is to have two source buffers on our Media Source: one for the video data, and the other for the audio

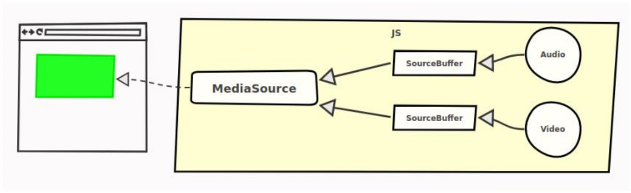


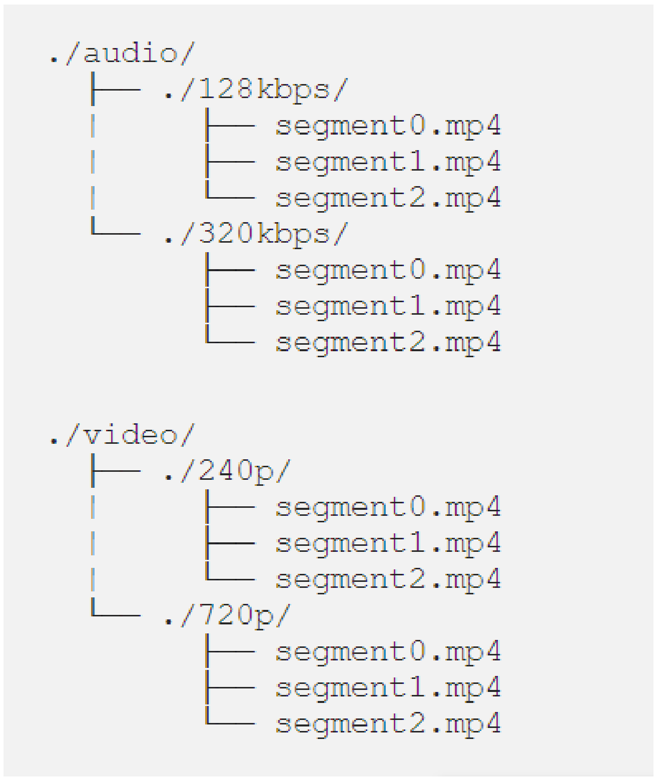
Fig 8

4.2.3 Media Segments

In advanced video players, is that video and audio data are split into multiple "segments". These segments can come in various sizes, but they often represent between 2 to 10 seconds of content. All these video/audio segments then form the complete video/audio content. Those "chunks" of data add a whole new level of flexibility, instead of pushing the whole content at once, we can just push progressively multiple segments. This means that we also have those multiple segments on server-side. From the previous example, our server contains the files as shown in fig. 9.

4.2.4 Adaptive Streaming

It is a feature, where the quality is automatically chosen depending on the user's network and processing capabilities. This is a central concern of a web player called adaptive streaming. This behaviour is enabled because of the concept of Media Segments. On the server-side, the segments are actually encoded in multiple qualities. A web player will then automatically choose the right segments to download as the network or CPU conditions change.(fig. 9)



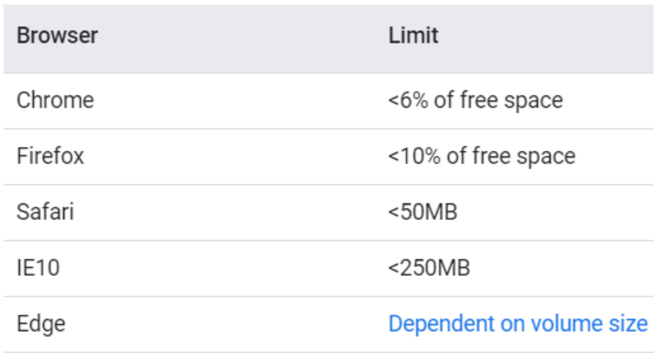
```
./audio/
├── ./128kbps/
|       ├── segment0.mp4
|       ├── segment1.mp4
|       └── segment2.mp4
└── ./320kbps/
        ├── segment0.mp4
        ├── segment1.mp4
        └── segment2.mp4

./video/
├── ./240p/
|       ├── segment0.mp4
|       ├── segment1.mp4
|       └── segment2.mp4
└── ./720p/
        ├── segment0.mp4
        ├── segment1.mp4
        └── segment2.mp4
```

Fig. 9

4.2.5 Offline Storage

Internet connections can be non-existent on the go, which is why offline support and reliable performance are common features in progressive web apps. Even in perfect wireless environments, judicious use of caching and other storage techniques can substantially improve the user experience. For the network resources necessary to load your app while offline, we used the Cache API (part of service workers). For PWAs, you can cache static resources, composing your application shell (JS/CSS/HTML files) using the Cache API and fill in the offline page data from IndexedDB. Debugging support for IndexedDB is now available in Chrome (Application tab), Opera, Firefox (Storage Inspector) and Safari .



Browser	Limit
Chrome	<6% of free space
Firefox	<10% of free space
Safari	<50MB
IE10	<250MB
Edge	Dependent on volume size

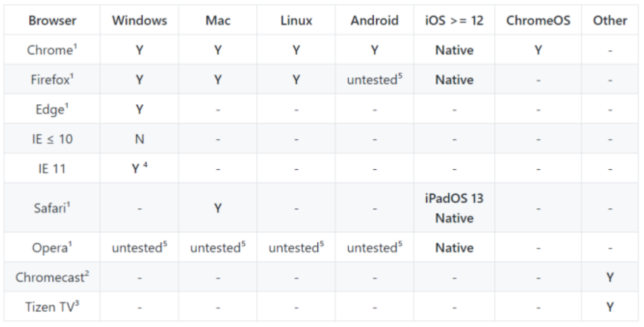
Fig 10 : Storage limits for browsers

4.3 Media Player

The choice of player for the project involved discussion on various factors and team concluded to opt for Shaka player because of it's wide range of features and libraries available to implement in an efficient and easy manner.

4.3.1 Shaka Player

Shaka Player is an open-source JavaScript library for adaptive media. It plays adaptive media formats (such as DASH and HLS) in a browser, without using plugins or Flash. Instead, Shaka Player uses the open web standards Media Source Extensions and Encrypted Media Extensions. Shaka Player also supports offline storage and playback of media using Indexed DB. Content can be stored on any browser. Storage of licenses depends on browser support. It involves support for multiple browsers. (fig. 10) and also have support for DASH and HLS manifest.



Browser	Windows	Mac	Linux	Android	iOS >= 12	ChromeOS	Other
Chrome[1]	Y	Y	Y	Y	Native	Y	-
Firefox[1]	Y	Y	Y	untested[5]	Native	-	-
Edge[1]	Y	-	-	-	-	-	-
IE ≤ 10	N	-	-	-	-	-	-
IE 11	Y [4]	-	-	-	-	-	-
Safari[1]	-	Y	-	-	iPadOS 13 Native	-	-
Opera[1]	untested[5]	untested[5]	untested[5]	untested[5]	Native	-	-
Chromecast[2]	-	-	-	-	-	-	Y
Tizen TV[3]	-	-	-	-	-	-	Y

Fig. 11

5. Results

Team was able to achieve the requirements with all the functionalities expected and below mentioned are our findings

5.1 UI/UX

The various elements as decided in the wireframing were implemented with multiple windows and components. The glimpse of some windows is shown in Fig. 12

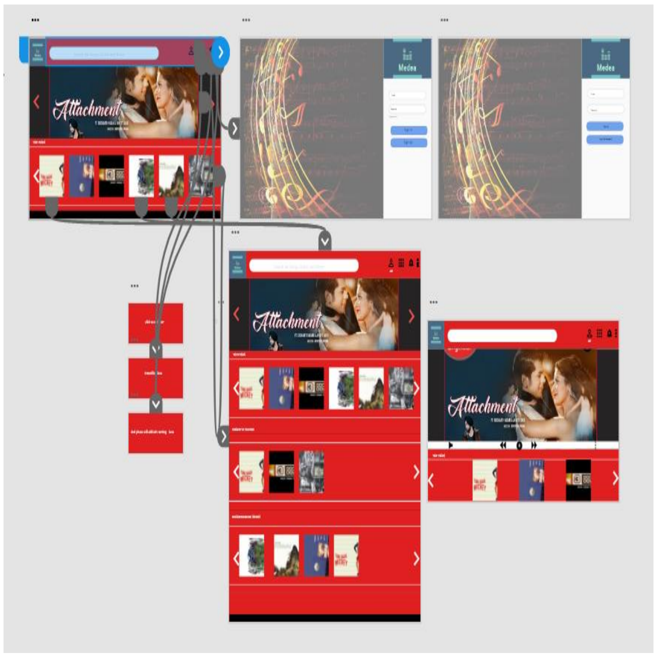
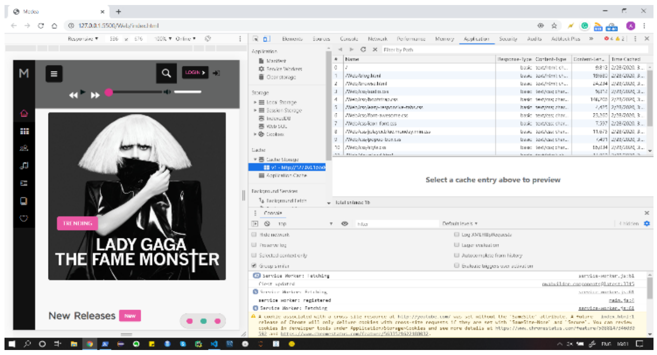


Fig 12 : Wireframing of the application

5.2 Offline playback

The offline playback functionality is achieved by caching the UI elements and some videos to cope up with the down network conditions.Fig. 13 depicts the basic UI elements loaded in absence of internet



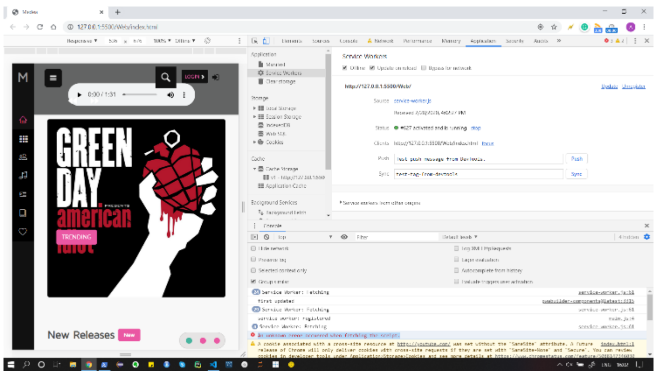


Fig. 13

5.3 Download Media

The in-app download functionality is integrated along the player so that user can download the media on application, which is cached in backend and can be played while being offline. Glimpse of Download feature can be seen in Fig. 14

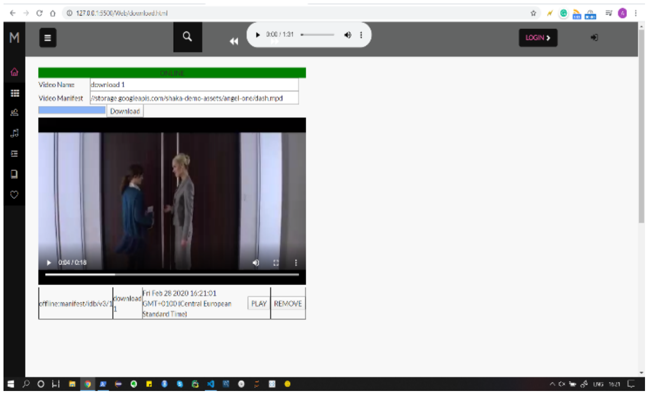


Fig. 14

5.4 Lighthouse Testing

Lighthouse is an open-source, automated tool for improving the quality of web pages. You can run it against any web page, public or requiring authentication. It has audits for performance, accessibility, progressive web apps, SEO and more. Lighthouse in Chrome DevTools, from the command line, or as a Node module. You give Lighthouse a URL to audit, it runs a series of audits against the page, and then it generates a report on how well the page did. From there, use the failing audits as indicators on how to improve the page. Each audit has a reference doc explaining why the audit is important, as well as how to fix it..

Lighthouse Test results were positive for our application and is qualified to be called as a Progressive Web Application. Test results are shown in Fig. 15

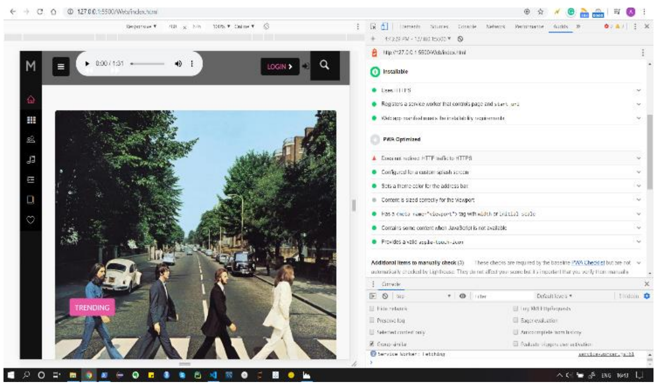
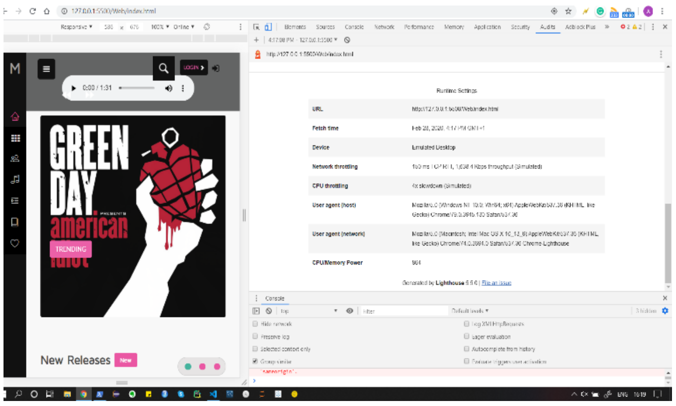


Fig. 15

5.5 Browser Testing

The application has been tested in multiple browsers to check if functionalities doesn’t fail for different browsers. The application ran successfully on , . The test results can be seen in Fig. 16



Runtime Settings

URL	http://127.0.0.1:5500/Web/index.html
Fetch time	Feb 28, 2020, 4:17 PM GMT+1
Device	Emulated Desktop
Network throttling	150 ms TCP RTT, 1,638.4 Kbps throughput (Simulated)
CPU throttling	4x slowdown (Simulated)
User agent (host)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36
User agent (network)	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3694.0 Safari/537.36 Chrome-Lighthouse
CPU/Memory Power	984

Generated by **Lighthouse** 5.5.0 | File an issue

Fig. 16

References

Github: **https://github.com/NautiyalAmit/Medea**

1. Ater, T. (2017). Building Progressive Web Apps: Bringing the Power of Native to the Browser. O'Reilly.
2. Ciman, M. and Gaggi, O. (2016). An empirical analysis of energy consumption of cross-platform frameworks for mobile development. Pervasive and Mobile Computing
3. Osmani, A. (2015). Getting started with progressive web appsK. Elissa, "Title of paper if known," unpublished.
4. Pedersen, M. (2016). Progressive web apps: Bridging the gap between web and mobile
5. Gaunt, M. (2016). Service Workers: an introduction.
6. Hume, D. A. (2017). Progressive Web Apps. Manning
7. Pedersen, M. (2016). Progressive web apps: Bridging the gap between web and mobile.