УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

Реализација система за синхронизовану аудио репродукцију

Вељко Селаковић

Ментор: проф. др Милош Цветановић
Модул Софтверско инжењерство
Електротехнички факултет
Београд, Србија, 2023

Реализација система за синхронизовану аудио репродукцију

Вељко Селаковић

Модул Софтверско инжењерство

Садржај

Глава 1

Увод

У савременом добу, дигитална репродукција аудио садржаја је постала неодвојив део нашег свакодневног живота. Од музике и подкаста до аудио књига и филмских садржаја, приступ квалитетном звуку је кључни елемент наше медијске конзумације. Са растом популарности стриминг платформи и апликација за аудио, потреба за ефикасном и поузданом репродукцијом никада није била већа.

Главни циљ овог рада је дизајнирати и имплементирати аудио систем који је способан за синхронизовани плејбек на више уређаја преко мреже. Ово је од суштинског значаја за кориснике који желе да уживају у заједничком искуству слушања музике или других аудио садржаја без обзира на своју географску локацију. Осим тога, овај пројекат упознаје читаоца са појмовима као што су *CDN*, *load balancing*, *WebSocket* и друго. Перформанс сервиса је тема која се такође обрађује, као и њен значај за евентуално скалирање у будућности.

У другом поглављу ће бити пружен преглед архитектуре система. Биће описан ток размишљања приликом пројектовања пројекта, као и разлози бирања појединих технологија.

У трећем поглављу биће описане технологије које су коришћење за реализацију клијентског и серверског дела апликације.

У четвртом поглављу ће бити приказани битни аспекти имплементације апликације уз објашњење њихове употребе. Такође, биће пружени примери кода који ће демонстрирати имплементацију битних делова приликом реализације.

У петом поглављу ће пре свега бити извучен закључак о реализованој апликацији. Поред тога биће предложена могућа проширења и унапређења система.

Систем развијен за овај дипломски рад омогућава корисницима да истовремено репродукују аудио садржај на различитим уређајима, обезбеђујући висококвалитетно искуство слушања без прекида или великог кашњења.

Глава 2

Архитектура система

У овом поглављу се говори о пројектовању самог пројекта. Један од примарних циљева овог дипломског рада је описивање разлика између развоја апликације са хоризонталним скалирањем на уму од класичне веб апликације.

2.1 Опис система

Како бисмо могли да конструишемо архитектуру за одређени систем, морамо знати које су нам функционалности неопходне. Име ове апликације је *Sync.mp3*. Идеја је да у апликацији постоје аудио собе на које људи могу да се повежу и заједно слушају шта год желе. Разлика у репродукцији за сваког корисника у једној соби треба да буде минимална.

У свакој соби постоје дељене контроле. Сви корисници су једнаки, свако може да паузира или промени активни аудио запис. Сваки корисник може да изађе из собе кад год пожели и уђе у неку другу. Свака аудио соба има своју плејлисту у којој корисници бирају активни аудио запис.

Тема коју обрађује рад, синхрона репродукција аудио фајлова је одлична подлога за компарацију различитих приступа скалирању. Прва ставка приликом пројектовања јесте локација складиштења свих аудио записа.

2.2 Чување аудио записа

Прва и логична помисао је да сви аудио записи могу бити складиштени на истом серверу на коме се налази и остатак апликације. Ово није добра идеја превасходно из разлога пто бисмо морали да алоцирамо сервере који имају довољно јаке спецификације да покрећу нашу апликацију и уз то опслужују медијски садржај кад год се затражи (енг. on-demand).

Следећа ствар на списку јесте алоцирати један цео сервер који ће функционисати као провајдер аудио захтева. Ово решава проблем загушења једног сервера, међутим, доста је скупље. Такође, заједнички проблем за обе идеје јесте испоручивање записа корисницима који могу да се налазе на два различита континента. Једном кориснику би време чекања (енг. latency) био у десетинама милисекунди, док би други чекао и десетине пута више.

CDN (Content Network Delivery) је мрежа рачунара разбијених по планети који имају идентичан садржај и служе за испоруку различитих пакета својим корисницима. Велики број интернерт платформи чији је велики удео саобраћаја медијски садржај (*Instagram*, *Facebook*, *Netflix*, ...) користе свој *CDN* како би сви корисници добијали приближно једнаку услугу, без обзира на то где се они налазе. Од зачетка овог концепта до данас, настало је много компанија чији је примарни позив пружање *CDN* услуга. У овом пројекту користићемо једну такву платформу.

2.3 RTMP или WebSocket?

Идеја синхронизоване репдрокуције садржаја није нова. Њену популарност можемо приметити на феномену *стриминга* (енг. streaming). Корисници користе доступне платформе за директан пренос видеа у скоро па реалном времену. Све веће *стриминг* платформе користе *RTMP* (Real-Time Messaging Protocol).

RTMP је идеалан за све пројекте где креатор има масивну публику која конзумира његов садржај. Кључна разлика између класичног *стриминга* и овог пројекта је равноправност свих учесника. Сваки учесник може да промени садржај, или да га паузира. Тестови спроведени пре почетка писања рада указали су на то да RTMP има упитне перформансе код честог паузирања садржаја. Дешавало се да се клијентима који се накаче пушта садржај од пре 30 секунди или више. Ово је прихватљиво за садржај какав се обично *стримује*, као што су вишечасовне сесије играња видео игара. Међутим, кашњење тог нивоа за на пример, песму која траје 2 до 3 минута је велик проблем.

Решење које се само наметнуло је коришћење протокола *WebSocket*. *WebSocket* омогућава комуникацију клијент-сервер у реалном времену. За разлику од *HTTP* (Hypertext Transfer Protocol), *WebSocket* је *full-duplex* и обе стране могу да шаљу поруке по потреби. Приликом тестирања, показало се да и *WebSocket* може да буде спор, у завсности од платофме на којој је имплементирана комуникација.

2.4 Избор технологије за серверски део

У случају пројекта као што је овај, потребно нам је окружење која подржава висок ниво паралелизације и брзо извршавање. Традиционално би се за апликацију оваквог типа узимало неко класично веб серверско окружење (*Node.js, PHP, Java*) које би креирало временски критичне таскове који се шаљу другим микросервисима на обраду. Микросервиси који обрађују захтеве високог приоритета где је време од критичног значаја су обично написани у неком од перформантнијих језика као што су *Go* или *Rust*.

Због тога што је једина временски критична ствар у овој апликацији имплементација *WebSocket* протокола, ово је идеална прилика да се тестира ново окружење, *Bun*. *Bun* обећава и до осам пута брже перформансе од *Node.js*, са опцијом да ради као *drop-in replacement* за *Node.js*.

Осим сервера који имплементира *WebSocket* протокол, креирају се и сервери који ће да обрађују класичне *HTTP* захтеве. Ови сервери су *stateless*, односно неће имати никакве податке о тренутном стању система. Ово нам омогућава да их скалирамо хоризонтално, односно да може да постоји више инстанци истог сервера који опслужује већи број корисника.

Када имамо више од једне инстанце, следећи корак је направити и *load balancer* који ће равномерно распоређивати захтеве на све сервере. *NGINX* пружа једноставан начин за креирање и конфигурисање оваквог система. *Round Robin* распоређивање је сасвим прикладно.

2.5 Избор базе података

Избор и архитектура базе падатака за било који пројекат је увек подухват сам по себи. Постоји велики број различитих технологија свака са својим предностима и манама. За апликацију коју креирамо, *MongoDB* је природан избор због лакоће интеграције са остатком система.

Проблем који *MongoDB* има у овој ситуацији је што спречава хоризонтално скалирање базе. Због тога што су сервери *stateless*, ту чувамо одређене информације које су потребне систему у реалном времену. *MongoDB* није *строго конзистентан* (енг. strong consistency), већ евентуално конзистентан (енг. eventual consistency). Када се *MongoDB* разбија на више инстанци, обично постоји једна инстанца која је *господар* (енг. master), док су друге *робови* (енг. slave). Све операције читања иду на робове, а све операције уписа на господара. Након уписа, господар шаље исти упис који је он добио свим робовима, и та пропагација траје одређено време. За то време може да пристигне захтев који ће прочитати невалидну вредност са једног од робова.

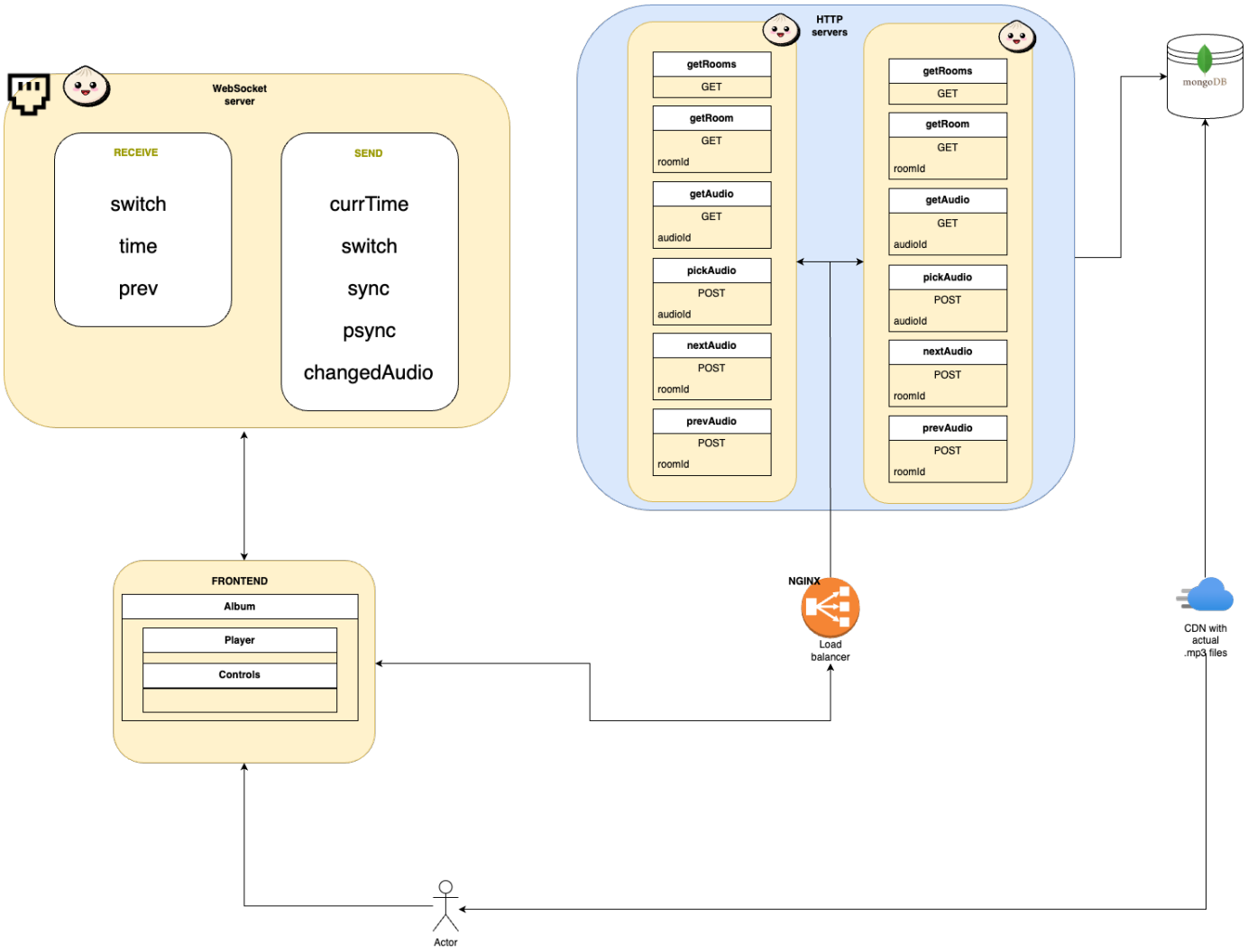
Скалирање *MongoDB* базе се углавном ради ако постоји изузетно велика

количина уписа у базу, реда више милиона, што овде није случај. Из тог разлога ће база бити само једна инстанца без могућности да се скалира на више инстанци.

Начин на који бисмо могли да укључимо и скалирање базе је ако пребацимо све елементе који су потребни у реалном времену у неки *in-memory* кеш, као што је *Redis*.

2.6 Избор окружења за клијентски део

React је један од најкоришћенијих библиотека за писање *SPA* (Single Page Application) сајтова. Једноставност релација компоненти и њихова хијерархија, као и доступност великог броја јавно доступних компоненти чине *React* очигледним избором. Циљна публика овакве апликације укључује и кориснике на телефонима, а *React* подржава прогрмирање респонзивних интерфејса сам по себи што је додатан бонус.



Слика 2.1: Дијаграм архитектуре система

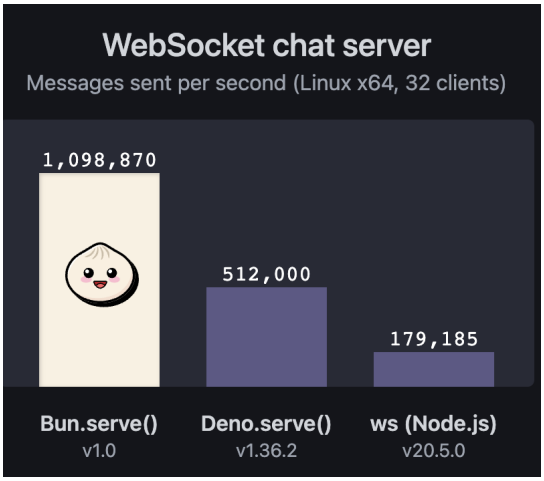
Глава 3

Опис коришћених технологија

У овом делу ће бити описане технологије које су коришћене приликом реализације пројекта, као и резоновање због којих су изабране.

3.1 Bun

Bun је ново окружење (енг. runtime) за *JavaScript/TypeScript*. Изабран је због одличне комбинације лакоће коришћења и перформанси. Пружа једнаку једноставност као и *Node.js*, са неколико пута бољим перформансама. *Bun* се у овом пројекту користи искључиво за *WebSocket* комуникацију. Преостали дијалог клијент-сервер се извршава преко класичне комбианције *Node.js* и *Express*



Слика 3.1: Поређење перформанси

3.2 Node.js

Познат по својој једноставности и брзини развоја, *Node.js* је један од најкоришћенијих платформи за развој веб апликација. Још једна његова предност је што може да се користи и на клијентској и на серверској страни што може бити значајно приликом развоја. Један програмер може направити комплетну апликацију познавајући само *JavaScript/TypeScript*.

3.3 Express.js

Express.js је *open-source* радни оквир за израду апликацију који се користи са *Bun* окружењем. *Express.js* служи за једноставно управљање *HTTP* захтевима које клијентска апликација шаље серверу. Осим брзог и ефикасног креирања путања за све захтеве, *Express.js* подржава тзв. програме средњег слоја (енг. middleware). Због тога је јако лако додати неке напредније опције попут контроле *CORS* (Cross-origin resource sharing) и лимитирање броја захтева од стране једног клијента.

3.4 MongoDB

MongoDB је најкоришћенији пример *NoSQL* базе података. Подаци се чувају као документи у склопу једне колекције. *MongoDB* пружа велику флексибилност и могућност ефикасног скалирања. Слабост овог система је недостатак референцијалног интегритета који је основна особина релационих база података.

3.5 NGINX

NGINX је *open source* веб сервер који осим основне намене може да се користи као *reverse proxy*, *load balancer*, *mail proxy* или као сервис за кеширање *HTTP*. У овом случају, *NGINX* је конфигурисан као *load balancer* са *round robin* методом избора.

3.6 React

Ова клијентска библиотека за *JavaScript/TypeScript* је једна од најкоришћенијих на свету. Омогућава ефиаксан и брз развој корисничког интерфејса. Главна карактеристика *React.js* је виртуелни DOM (Document Object Model) који омогућава ефикасно ажурирање само делова веб странице који су се променили, без потребе за целокупним учитавањем странице.

3.7 CDN (Content Delivery Network)

Ефективно репродуковање аудио фајлова захтева брзо учитавање у меморију. Одзив ка серверу који се налази преко Атлантског океана није ни близу одзиву сервера који се налази у Франкфурту. Из тог разлога су овде коришћени сервиси *Bunny.net* као бесплатна алтернатива великим сервисима као што су *Fastly* или *Cloudflare*. Једино што складиштимо на њиховим серверима су *.mp3* датотеке који ће се учитавати са клијентске стране.

3.8 Docker

Docker је платформа која омогућава развој, паковање и извршавање апликација у контејнерима. Контејнери су изоловани системи који функционишу независно од оперативног система који их покреће, налик на виртуелне машине. Контејнеризација апликације омогућава њено конзистентно понашање, као и лакоћу дупликације и померања на друге сервере.

Глава 4

Имплементација система

Систем је реализвоан у три корака:

1. Креирање контејнера који управља над *WebSocket* протоколом
2. Имплементација традиционалног *HTTP* сервера са *MongoDB* bazom podataka
3. Развијање клијентског дела и повезивање са серверским делом

Опциони четврти корак је *deployment* апликације на продукцијски сервер.

4.1 Реализација WebSocket протокола

4.1.1 Речник у комуникацији *WebSocket*-клијент

Како бисмо остварили ефикасну комуникацију између клијента и сервера, морамо креирати и језик којим ће комуницирати. Даље су наведени коришћени појмови са објашњењем.

- **currTime** (сервер → клијент) - Сервер пита клијенте докле су стигли са аудио записом
- **switch** (сервер → клијент) - Сервер каже клијенту да промени статус репродукције (паузирати или пустити даље)
- **sync@x.xx** (сервер → клијент) - Сервер каже клијенту да синхронизује своју репдроукцију на дато време
- **psync@x.xx** (сервер → клијент) - Сервер каже клијентима да синхронизује своју репродукцију на дато време, али да остане паузиран

- **changedAudio** (сервер → клијент) - Сервер каже клијентима да је промењен активни аудио запис
- **switch** (клијент → сервер) - Клијент каже серверу да проследи осталима да промене статус репродукције
- **time@x.xx** (клијент → сервер) - Клијент одговара серверу докле је стигао са аудио записом
- **next** (клијент → сервер) - Клијент каже серверу да је променио активни аудио запис на следећи са плејлисте и да јави осталим клијентима.
- **prev** (клијент → сервер) - Клијент каже серверу да је променио активни аудио запис на прошли са плејлисте и да јави осталим клијентима.

4.1.2 Иницијализација контакта и затварање

```
const server = Bun.serve<{ username: string, room: string }>({
  port: 5000,
  fetch(req, server) {
    let _url = new URL(req.url);

    const success = server.upgrade(req,{
        data: { username: _url.searchParams.get("username"),
                room: _url.searchParams.get("room") }
      });
    if (success) {
      // Bun automatically returns a 101 Switching Protocols
      // if the upgrade succeeds
      return undefined;
    }

    // handle HTTP request normally
    return new Response("sync.mp3!");
  },
  websocket: {
    message: receiveMessage,
    open: openSocket,
    close: closeSocket, // a socket is closed
    drain(ws) {},
    publishToSelf: true
  },
});
```

На серверском делу креирамо објекат који ће прихватати и обрадити све постојеће и надолазеће захтеве. Следеће *callback* функције су биле потребне:

- *message* - Обрађивање поруке од стране клијента
- *open* - Обрађивање захтева за отварањем нове конекције
- *close* - Обрађивање захтева за затварањем једне од конекција

Обрада отварања конекције

```
const openSocket = function (ws: ServerWebSocket<{ username: string, room: string }>) {
  console.log(ws.data.room);
  ws.subscribe("music" + ws.data.room);

  console.log("Opening a socket!")
  console.log("Username: " + ws.data.username);

  if (ws.isSubscribed("music" + ws.data.room) ) {
    console.log("successfully subscribed to " + "music" + ws.data.room);
    server.publish("music" + ws.data.room, `${ws.data.username} just joined the channel`);
  }

  server.publish("music" + ws.data.room, "currTime");
}
```

Код за отварање је једноставан, узимамо собу која је тражена и отворену конекцију уписујемо као део *канала* (енг. channel). Након овога, све што се објави (енг. publish) на тај канал стиже директно клијенту као порука. Одмах након отварања конекције, шаљемо поруку *currTime* осталим клијентима на истом каналу како бисмо видели докле су стигли са извршавањем аудио записа.

```
if (ws === null) {
  const socket = new WebSocket("ws://157.230.104.92:83/play?username=" + usr + "&room=" + id.toString());
  setWs(socket);
}
```

Са клијентске стране отварамо конекцију када је време за то и дајемо 2 параметра: корисничко име и јединствени идентификатор собе на коју желимо да се повежемо.

Обрада затварања конекције

Затварање конекције је и са серверске и са клијентске стране кратко решена. Код серверског затварања битно је да се конекција избаци из њеног канала. Клијентско затварање се дешава након што се одради *unmount* компопненте.

```
const closeSocket = function(ws: ServerWebSocket<{ username: string, room: string }>) {
  const msg = `${ws.data.username} has left`;
      ws.unsubscribe("music" + ws.data.room);
      server.publish("music" + ws.data.room, msg);
}
```

Слика 4.1: Серверско затварање конекције

```
useEffect( () => () => {
  console.log("unmount");
  console.log(ws);
  if (ws === null) return;
  ws.close();
}, [] );
```

Слика 4.2: Клијентско затварање конекције

4.1.3 Комуникација клијент - сервер

Сва комуникација клијент - сервер иде преко већ поменутих команди. Команде се парсирају и одговарајући одговор се шаље. У суштини, команде можемо поделити на 3 категорије:

- Команде повезивања са аудио собом
- Команде мењања аудио записа
- Команде мењања статуса репродкуције

Синхронизација аудио записа

Повезивање са аудио собом је већ представљено у делу отварања конекције. Након успешно отворене конекције, сервер шаље свим клијентима команду **currTime**. Клијенти шаљу одговор користећи команду **time**. Додатна информација коју клијенти могу да пошаљу је паузирани статус репродукције. У том случају команда узима следећи формат: **time@-1@x.xx**, где други параметар представља докле је клијент стигао са репродукцијом. У зависносто од тога да ли су клијенти паузирани или не, сервер шаље назад команду **sync** иил **psync** са временом на које се сви синхронизују.

Мењање аудио записа

Мењање аудио записа се започиње корисничком акцијом промене песме. Клијент тада шаље команде **next** иил **prev**. Пре слања команде се уради позив ка *HTTP* серверу да промени индекс тренутно активног аудио записа у бази. Због тога знамо да је, у моменту кад сервер прими команде, већ промењено стање у бази. Једина дужност сервера у том моменту је да јави свим осталим клијентима да је активни аудио запис промењен командом **changedAudio**.

Сви клијенти након пријема ове поруке раде поновно учитавање записа и чекају команду за репродукцију.

Промена статус репродукције

Најједноставнија категорија од три наведене. Започиење се акцијом корисника којом мења статус репродукције аудио записа на паузиран или пуштен. Одмах након тога се шаље команда **switch** ка серверу који само пропагира истоимену команду назад свим клијентима. Када клијент прими команду **switch**, одмах мења свој статус репродукције у супротни.

4.2 HTTP сервер и база података

4.2.1 MongoDB колекције

Користимо *MongoDB* као главну базу података. Имамо две релевантне колекције: *rooms*, и *audios*. Прва колекција садржи информације о аудио собама, и има колекцију идентификатора аудио записа у плејлисти. Колекција *audios* садржи све информације о аудио документу, укључујући и директан линк до *.mp3* датотеке на *bunny.net* сервису који се понаша као *CDN*.


```
  _id: ObjectId('65098e67fb32220bd8d43d55')
  id: 0
  name: "Random room"
  description: "A cool room with random songs"
  image: "https://source.unsplash.com/random?wallpapers"
  audio: "https://file-examples.com/storage/fe9315700c650841d9ee30d/2017/11/file…"
}playlist: Array (6)
    0: 1
    1: 2
    2: 3
    3: 4
    4: 5
    5: 6
  activeAudio: 1
  __v: 0
```


Слика 4.3: Пример аудио собе

Најбитнија поља ових докумената су *activeAudio* и *playlist*. Поље које садржи плејлисту садржи листу јединствених идентификатора песама преко којих можемо да приступимо њиховим документима у другој колекцији. Поље *activeAudio* садржи индекс тренутно активног аудио записа у низу *playlist*.

```
_id: ObjectId('6509845d649600549312d452')
d: 1
name: "Danielle (smile on my face)"
artist: "Fred Again"
image: "https://source.unsplash.com/random?wallpapers"
audio: "https://sync.b-cdn.net/content/Fred%20again..%20-%20Danielle%20(smile%…"
__v: 0
```


Слика 4.4: Пример аудио записа

Код докумената аудио записа најрелевантније поље јесте *audio*. *Audio* поље садржи директан линк ка *bunny.net* хостованом *.mp3* фајлу. Овиме избегавамо коришћење додатног медија сервера, и додатно повећавамо перформансе система.

4.2.2 Поставка Express.js система

Код подешавања *Express.js* радног оквира можемо издвојити две целине:

- Повезивање са базом података користећи *ORM* (Object-relational mapping)
- Исписивање пословне логике за све тражене захтеве

Повезивање базе података са *Express.js* радним оквиром

Користећи библиотеку *mongoose*, врло лако можемо повезати контејнер који покреће *MongoDB* са нашим серверским контејнером. Осим тога, креираћемо и објекте шема специфично за релевантна поља из наших докумената.

Имплементација обраде *HTTP* захтева

Имплементирани су следеће путање:

- **getRooms** - *GET* - враћа листу свих соба
- **getRoom** - *GET* - враћа информације конкретне собе
- **getAudio** - *GET* - враћа информације о аудио запису
- **pickAudio** - *POST* - шаље редни број песме у плејлисти која се пушта ван реда
- **nextAudio** - *POST* - захтев да се пусти следећа песма
- **prevAudio** - *POST* - захтев да се пусти прошла песма

```
const audioSchema = new mongoose.Schema({
    id: {
        type: Number,
        required: true
    },
    name: String,
    artist: String,
    image: String,
    audio: {
       type: String,
       required: true
    }
});
```

Слика 4.5: Пример шеме декларисан користећи *mongoose*

- **populate** - *POST* - техничка путања која популише базу у случају да не постоје документи

Имплементација већине метода који су наведени је слична, налик на остале *CRUD* (Create Read Update Delete) операције. Један тип путање се издваја: промена аудио записа. Два питања се намећу приликом имплементације:

- Која је употреба ових захтева, зашто не иде све преко *WebSocket* протокола?
- Како ћемо се заштити од злоупотребе?

На почетку смо донели одлуку да наши *HTTP* сервери буду *stateless* како бисмо могли лакше да скалирамо хоризонтално. Последица тога је да не можемо чувати икакве варијабле на нашем серверу због тога што у реалности постоје два или више истих сервера од којих ће сваки имати своју инстанцу променљиве. Свакако морамо памтити тренутно стање за сваку аудио собу, тако да смо за то једноставно додали као ново поље у документ сваке собе. Сваки сервер контактира исту базу и неће долазити до различитих закључака у односу на друге. Потенцијална алтернативно алтернативно решење може бити коришћење брже базе података, као што је на пример *Redis*, који је *in-memory*, што му даје значајну предност у брзини читања, писања и тражења у односу на конкурентне технологије. Велики недостатак *in-memory* база је веома ограничен капацитет.

Код система где су сви корисници равноправни, злоупотреба је реалан проблем. Малициозни корисници се тешко елиминишу али им се увек може отежати посао. У овом раду је имплементиран тзв. *rate limit* за промену аудио записа.

```
const nextAudioLimiter = rateLimit({
  windowMs: 10000, // 10s
  limit: 1,
  message:  'Too many requests, please try again later',
  keyGenerator: (req: any, res: any) => true
})
```

Слика 4.6: Пример опција за лимитирање слања захтева

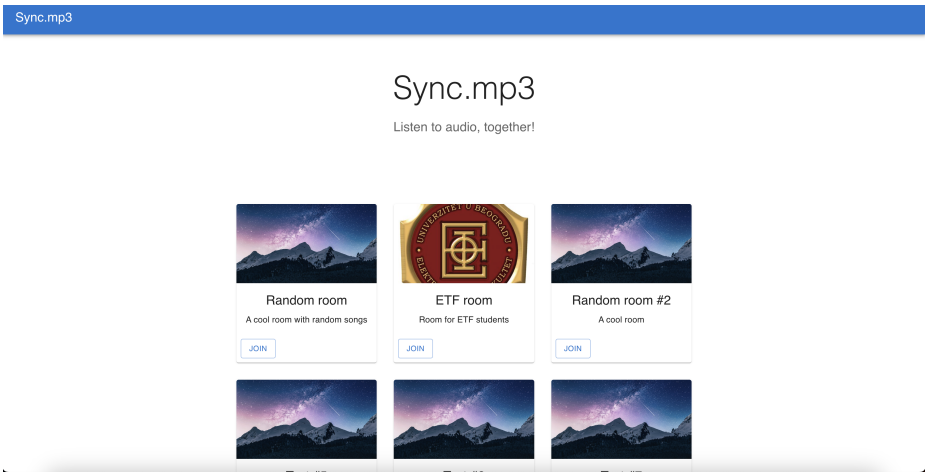
У периоду од 10 секунди, сваком кориснику је дозвољено да само једном промени аудио запис. Уз поруку, кориснику се враћа статус 429, познатији као *Too many requests*.

4.3 Израда клијентске апликације

Клијентска апликација се састоји из две велике целине. Прва целина је интерфејс у којем корисник одлучује у коју собу жели да уђе. Друга целина је сама соба, односно шта корисник види када уђе у собу. Већина подкомпоненти које су коришћене су део стандардне *MUI* (Material UI) библиотеке.

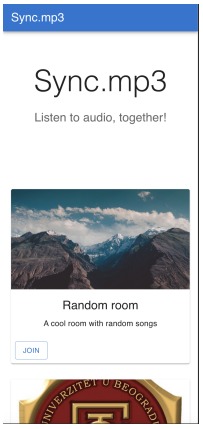
4.3.1 Приказ свих соба

Свака соба је приказана као *Card* компонента, која је део *MUI* библиотеке. Једини позив ка сервер из овог дела интерфејса је **getRooms** захтев који врати списак свих соба и основне информације о њима.



Слика 4.7: Приказ свих доступних соба

Један од приоритета апликације је такође и да омогући коришћење сервиа свим корисницима, са којег год уређаја они долазили. Због тога су све компоненте имплементиране са респонзивним дизајном на уму.

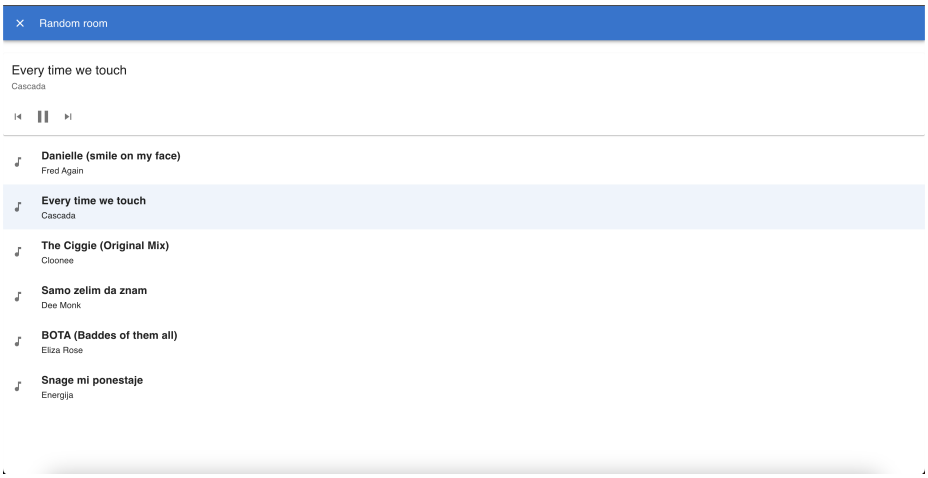


Слика 4.8: Приказ свих доступних соба на телефону *iPhone 12 Pro*

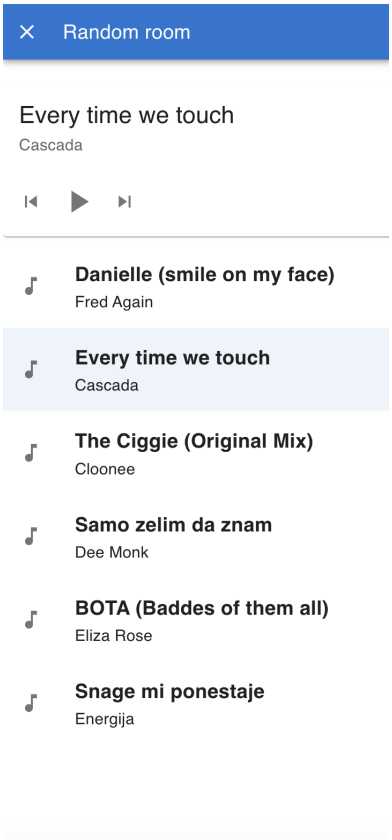
4.3.2 Интерфејс аудио собе

Компонента која контролише аудио плејбек је срж клијентске апликације. Инстанцирање везе са сервером преко *WebSocket* протокола се ради када се отвори сама соба. На аналогни начин се конекција затвара приликом гашења конекције ка целом сајту или једноставног изласка из собе који затвара цео контролни интерфејс.

Следеће две компоненте играју централну улогу у аудио соби: *Player* и њена подкомпонента *Controler*.



Слика 4.9: Изглед аудио собе



Слика 4.10: Изглед аудио собе на телефону *iPhone 12 Pro*

Контролер подкомпонента је у суштини само картица која садржи име и аутора тренутног аудио записа са додатним дугмићима који служе за контролу. *Player* компонента заузима цео дисплеј и садржи име собе, контролер као и интерактивну плејлисту са назначеним активним аудио записом. Корисник може да мења активни аудио запис користећи *напред* и *назад* дугмад као и кликом на било који елемент у плејлисти.

Контролер компонента шаље четири различита захтева ка серверу. Ти захтеви су редом:

- **getRoom** - Главна информација која се извлачи је активни аудио запис и плејлиста
- **getAudio** - Извлачење информације о записима
- **prevAudio** - Промени аудио запис на претходни
- **prevAudio** - Промени аудио запис на следећи

Player компонента додатно шаље и **pickAudio**, када корисник кликом на ред у плејлисти изабере нови активни аудио запис

```
<AppBar sx={{ position: 'relative' }}>
  <Toolbar>
    <IconButton
      edge="start"
      color="inherit"
      onClick={handleClose}
      aria-label="close"
    >
      <CloseIcon />
    </IconButton>
    <Typography sx={{ ml: 2, flex: 1 }} variant="h6" component="div">
      {currentRoom.name}
    </Typography>
  </Toolbar>
</AppBar>
<Controls id={id} fetchCallback={setCurrentSong} currSong={currentSong}/>
<List component="nav" aria-label="">
    {audios.map((audio) => (
      <ListItemButton key = {audio.id} onClick={() => pickAudio(audio.id)}
        selected={audio.id === currentSong}
      >
        <ListItemIcon>
          <MusicNoteIcon/>
        </ListItemIcon>
        <ListItemText disableTypography
            primary=
            {<Typography sx={{fontWeight: 'bold'}} variant="h6" style={{ color: '#ff' }}>{audio.name}</Typography>}
            secondary={audio.artist} />

      </ListItemButton>

      ))
    }
    </List>
```

Слика 4.11: Хијерархија подкомпоненти у *Player* компоненти

4.4 Deployment

Deployment је процес дистрибуције апликације или система на циљане рачунаре (у овом случају, *DigitalOcean droplet*), које ће користити крајњи корисници. Цео процес се може разложити на следеће независне кораке:

- Контејнеризација целог система
- Изнајмљивање сервера
- Пребацивање и активирање система на серверу

4.4.1 Контејнеризација система

За контејнеризацију система се користи *Docker* са *Docker compose* алатом. *Docker compose* користимо како бисмо оркестрирали све подапликације и њихове односе. Сваки контејнер са апликацијом може да комуницира са осталинма. Постоји 6 контејнера:

1. Клијентска аплиакција
2. *HTTP* сервер број 1
3. *HTTP* сервер број 2
4. *NGINX* инстанца која имплементира *load balancer*
5. *WebSocket* сервер
6. *MongoDB* инстанца

Користимо фајл *compose.yml* за конфигурацију система.

Конфигурација клијентске апликације

Наша клијентска апликација је писана у библиотеци *React*, тако да за основни *image* узимамо последњу верзију *Node.js* базирану на *Alpine* оперативном систему. Пребацујемо сав код командом *COPY*, као и фајл *package.json* у коме се налазе све релевентне библиотеке које користимо као и команде за покретање. Након што се *image* иницијализује, покрећемо *npm run build*, који компајлира наš *JavaScript/TypeScript* код. После тога командом *npm start* запокрећемо извршавање.

Конфигурације *HTTP* сервера

Користимо *Bun* као основу за нашу апликацију. Процес је сличан процесу за клијентску апликацију. Копирамо код и *package.json* фајл а затим користимо команду *bun run*. За развојне варијанте контејнера користимо *bun run hot* који ради освежавање сервиса приликом сваке промене у коду. У *compose.yml* фајлу имамо две инсганце ове апликације. Једина разлика је у порту на који их мапирамо. Касније се те информације прослеђују у *load balancer*, како би он знао да их преусмери на праву локацију.

Конфигурација *WebSocket* сервера

Иако је можда логички најкомплекснија апликација, овај контејнер се не разликује од претходних. Једина већа разлика између *Node* и *bun* база је у основном оперативном систему. *Node* користи *Alpine* док је *bun* на *mini Debian* оперативном систему.

Конфигурација *MongoDB* kontejnera

У овом случају не користи се ни једна од додатних опција, чак се и сам контејнер мапира на подразумевани порт 27017. Подаци се чувају локално јер је за контејнер мапиран *volume* на главном оперативном систему.

Конфигурација система за *load balancing*

За базу овог контејнера користимо *NGINX* на *Alpine* оперативном систему. Мењамо *nginx.conf* фајл.

Оваква комбинација подешавања распоређује захтеве према серверима у *Round Robin* алгортиму. Осим овог алгоритма, постоји још неколико начина избора где који захтев заврши. Један од занимљивијих за овај пројекат је *Generic Hash*, који распоређује захтеве базирано на додатном параметру који се пошаље уз исти. Примена овога би била у хоризонталном скалирању више *WebSocket* сервера, како би све конекције једне собе биле ка истим инстанцама сервера.

```
upstream loadbalancer {
  server web1:5000;
  server web2:5000;
}

server {
  listen 80;
  server_name 127.0.0.1;
  location / {
    proxy_pass http://loadbalancer;
  }
}
```

Слика 4.12: Поставке *nginx.conf* фајла

```
services:
  web1:
    restart: on-failure
    build: ./web
    hostname: web1
    ports:
      - '81:5000'
    volumes:
      - ./web:/usr/src/app
  web2:
    restart: on-failure
    build: ./web
    hostname: web2
    ports:
      - '82:5000'
    volumes:
      - ./web:/usr/src/app
  server1:
    restart: on-failure
    build: ./server
    hostname: server1
    ports:
      - '83:5000'
    volumes:
      - ./server:/usr/src/app
  nginx:
    build: ./nginx
    ports:
    - '84:80'
    depends_on:
    - web1
    - web2
  mongodb:
    image: mongo:latest
    ports:
      - 27017:27017
    volumes:
      - mongodb_data_container:/data/db
  frontend:
    restart: on-failure
    build: ./client/music-app
    hostname: client
    ports:
      - '80:3000'
volumes:
  mongodb_data_container:
```

Слика 4.13: *compose.yml* датотека

4.4.2 Изнајмљивање хардвера

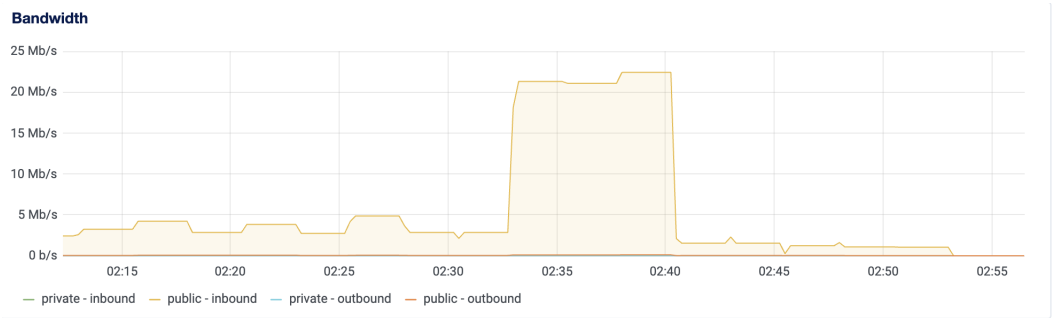
Као најбољи и најприступачнији приступ избацивања апликације у јавност се показао *DigitalOcean*. *DigitalOcean* је компанија чији су главни производи рачунарство у облаку и *IaaS* (Infrastructure as a Service). За овај пројекат нам је потребан само један *droplet* минималних спецификација.

Droplet је виртуелна машина базирана на Линукс оперативним системима која се покреће на виртуелизованом хардверу. Спецификације ове виртуелне машине су следеће:

- 1GB оперативне меморије
- 35GB складишне меморије на супер брзој *SSD* меморији
- 1 виртуелизовани процесор
- *Ubuntu 22.04* оперативни систем

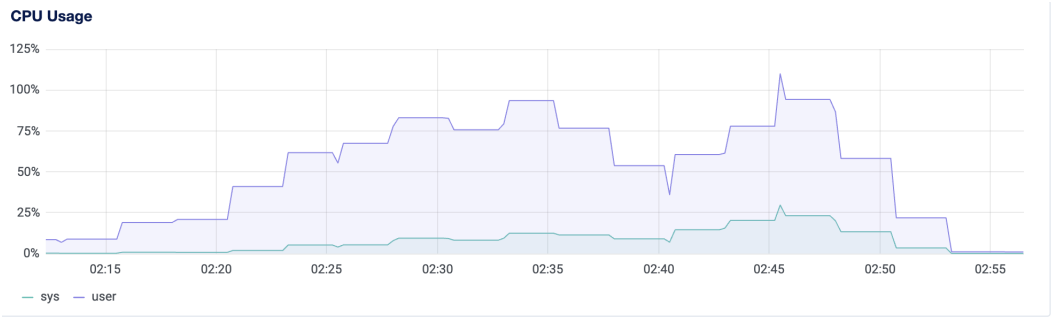
DigitalOcean нуди и вертикално и хоризонтално скалирање. Ако желимо да скалирамо вертикално, једноставно узмемо јачи *droplet*. Ако желимо да скалирамо хоризонтално узећемо додатне инстанце које су *droplet* истих капацитета.

Две веома битне могућности које *DigitalOcean* пружа су *SFTP* (Secure File Transfer Protocol) и детаљно бележење коришћења ресурса. *SFTP* је од велике важности због тога што омогућава трансфер великог броја фајлова без превеликог труда. На тај начин је сав извршни код пребачен на сервер. Помоћу графикона које можемо пратити у реалном времену може се проценити искоришћеност хардвера и проценити да ли је потребно даље се ширити или чак смањивати ако саобраћај опада.

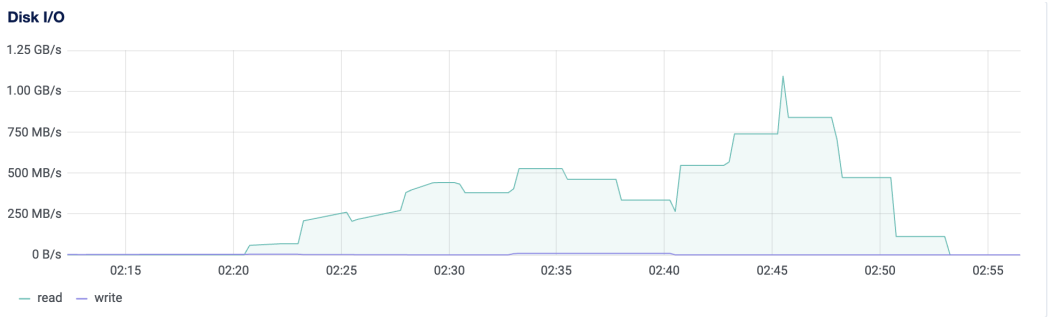


Слика 4.14: Графикон мрежног протока за време тестирања

Са ових графикона можемо приметити пораст свих параметара приликом тестирања када је у једном моменту било 5 корисника који активно користе апликацију.



Слика 4.15: Графикон искоришћења процесора за време тестирања



Слика 4.16: Графикон улазно-изланих операција на диску за време тестирања

4.4.3 Конфигурисање система на продукционом серверу

Прва ствар коју треба урадити је пребацити све релевантне датотеке за креирање система на продукцијски сервер. У овом случају то је одрађено преко *SFTP*, користећи креденцијале које нам је дао *DigitalOcean*. Након што је цела структура пребачена покрећемо команду *docker compose build* која креира све потребне *слике* (енг. image). Све што је преостало након тога јесте *docker compose up* који покрене цео пројекат са свим својим контејнерима. Опциони параметар који се користи на продукционом серверу је *-d*, који покреће у *detached* режиму. *Detached* режим је налик на *daemon* нит, покренут је у позадини без икаквог исписа на терминалу.

Глава 5

Закључци о систему и могућа унапређења

5.1 Унапређења

Иако систем функционише онако како је замишљено, остаје доста ствари које се могу проширити и побољшати.

5.1.1 Техничка унапређења

Различите могућнности унапређења су дискутоване у самом раду, али ће свакако бити поновљене и у овој секцији. Техничка унапређења се најпре односе на боље начине имплементације архитектуре и саме реализације пројекта.

На пример, ова апликација би имала бољи одзив кад би бележила мапу свих соба са тренутним аудио записом уместо радила *fetch* из базе. *Redis* би био идеалан за тако нешто.

Коришћење *Redis* платформе и као сервис за кеширање је такође сасвим валидан предлог. Свака елиминација непотребних упита ка бази може значајно убрзати респонзивност система.

Сам код има неколико места на којима би могао бити унапређен. Рецимо, команде *next* и *prev* које иду ка *WebSocket* машини су редудантне, може се креирати унификована команда за промену аудио записа.

5.1.2 Функционална унапређења

Осим техничке изведбе, битна је и функционалност апликације. Највеће унапређење платформе би дефинитивно био кориснички *upload* аудио фајлова на сервере. Ово међутим, осим финансијских, има и легалне последице ако корисници окаче аудио записе чији нису власници. У том случају би се морао

креирати посебан налог за креаторе који би качили само своја ауторска дела, као што је случај на свим већим аудио *стриминг* платформама као што су *Spotify*, *Deezer*, *Apple Music*, *Youtube Music* и остали.

Реаранжирање плејлиста би такође била изузетно корисна функционалност са доста јаким потенцијалом за злостављање од стране малициозних корисника. Функционалност *seek* је споменута од стране неколико тестера апликације. *Seek* представља могућност промене где се тренутно налазимо у аудио запису, односно бирање времена на које скачемо, а заједно са нама и сви који су у аудио соби. Као и реаранжирање плејлисте, и ова функционалност има изузетан потенцијал за непримерено понашање од стране разних корисника.

5.2 Коментар о пројекту

Као што је и на почетку написано, синхронизована репродукција аудио записа је изузетно добар пример на коме се може дискутовати на које све начине можемо скалирати наш систем како се број корисника повећава.

Сам пројекат је успешно реализован, тестиран и пуштен на продукцију где су могли да га користе и људи ван студентове локалне мреже. Утисци људио који су испробали апликацију су позитивни. Кашњење репродукције између уређаја је једва приметно иако су захтеви ишла од Београда до Франкфурта и назад.

Литература

1. Документација за React, https://react.dev/
2. Документација за Node.js, https://nodejs.org/
3. Документација за bun, https://bun.sh/docs
4. Документација за express.js, https://devdocs.io/express/
5. Документација за mongoose, https://mongoosejs.com/docs/
6. Документација за MongoDB, https://www.mongodb.com/docs/
7. Peer5, Setting up HLS live streaming server using NGINX, https://medium.com/@peer5/setting-up-hls-live-streaming-server-using-nginx-67f6b71758db
8. Martin Kleppmann, Designing Data-Intensive Applications, 2017