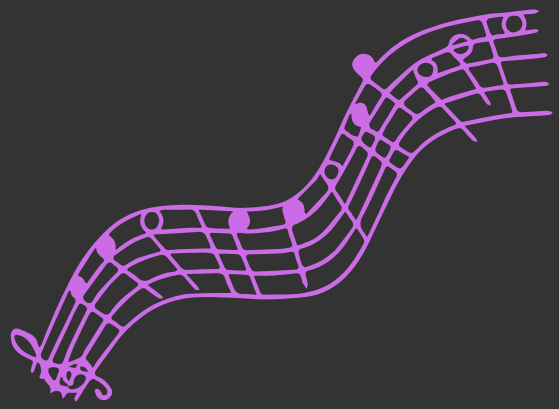


MUSIC PLAYER

PERSONAL PROJECT

서동현

INDEX

01 개요 ··· 03p~06p

02 화면설계 ··· 07p~10p

03 FRONT-END ··· 11p~17p

04 BACK-END ··· 18p~24p

05 후기 ··· 25p~26p

개요

프로젝트 개요

GO TO SITE!

개발 기간

2024.09.15 ~ 2024.09.22

- 화면설계 및 주요 기능 구현
- 웹서버 및 데이터베이스 구축

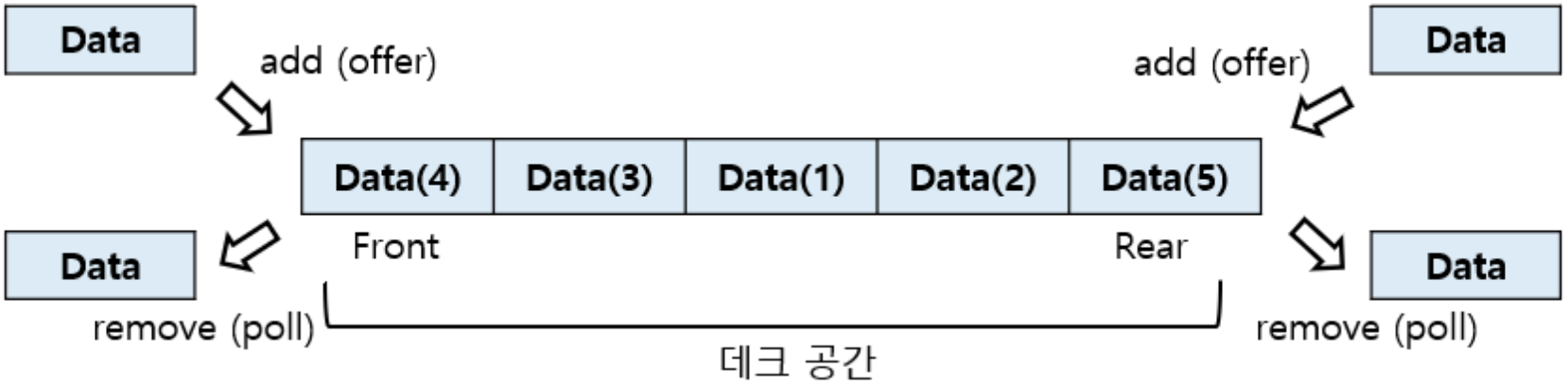
기술 스택

FRONT-END

BACK-END

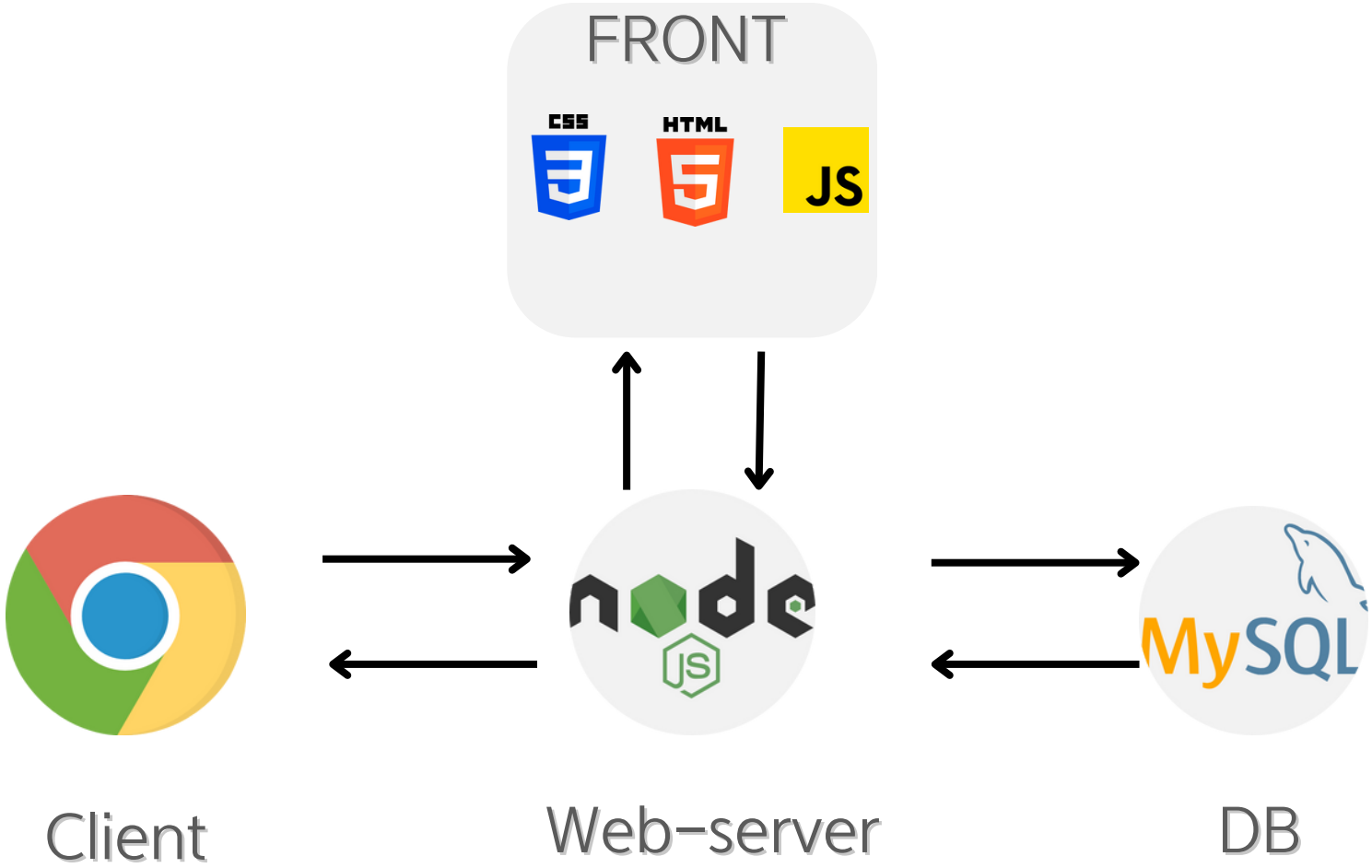
자료구조: DECK

GO TO SITE!



- 데크의 기존구조는 양방향 입출력이 가능한 구조
- push, pop, shift, unshift 연산을 실행할 수 있어 큐와 스택의 기능을 가지고 있음

- **덱(deque) 자료구조를 활용하여 음악 리스트를 저장하고, 한 방향으로 재생하는 순차 재생, 반대 방향으로 재생하는 역순 재생, 그리고 무작위로 재생하는 랜덤 재생 기능을 구현하였습니다.**

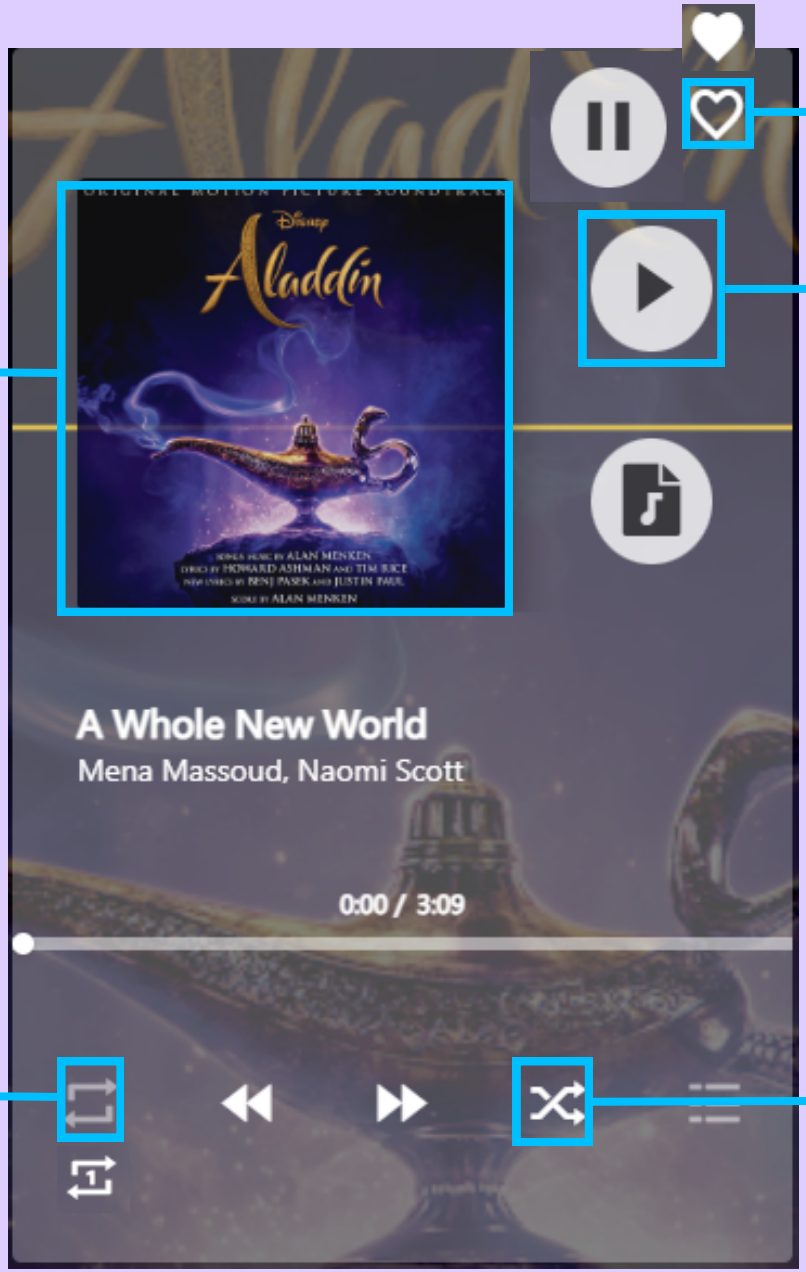

FRONT
CSS
HTML
JS
node
JS
MySQL
Client
Web-server
DB

GO TO SITE!

● 화면설계

화면구성

GO TO SITE!



재생되는 노래에 맞는 이미지를 출력

좋아요 버튼
클릭 시 데이터베이스 저장

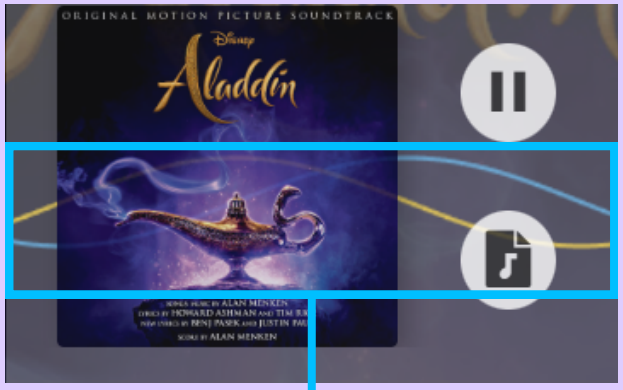
음악 재생 버튼

반복 재생 버튼

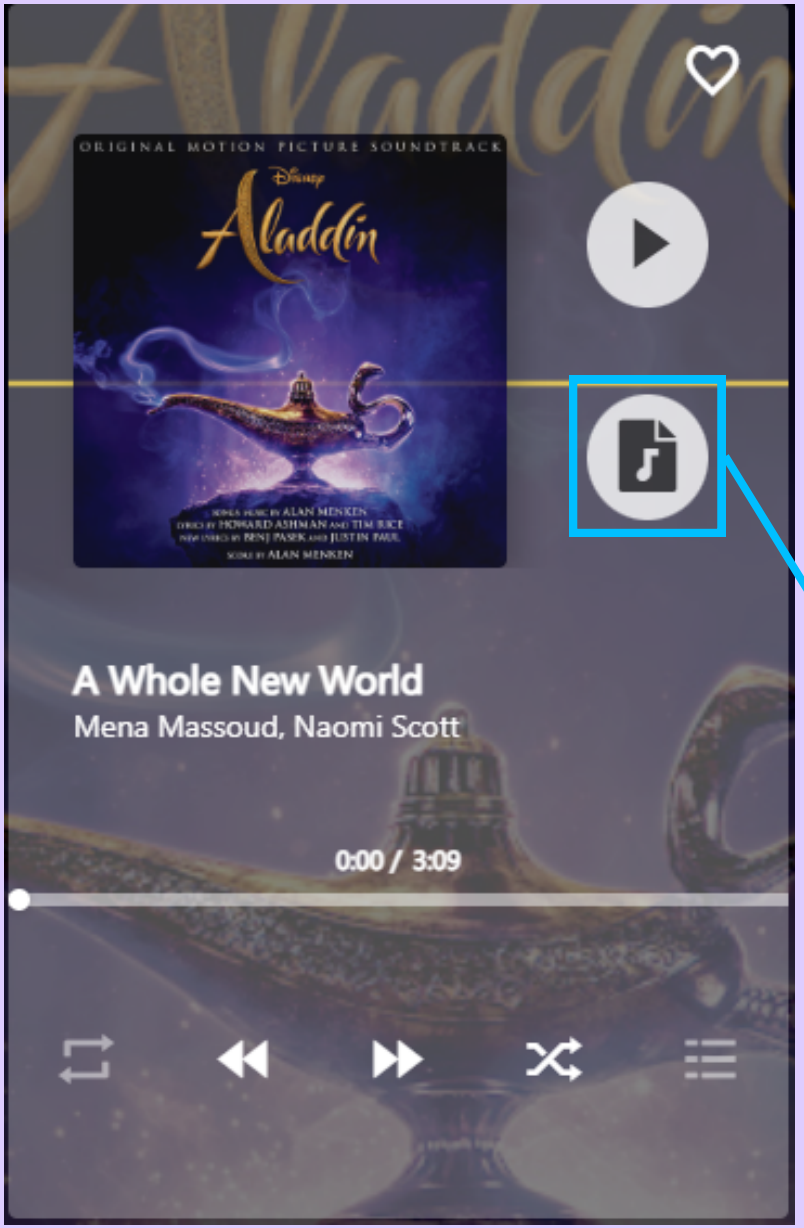
랜덤 노래 재생 버튼

화면구성

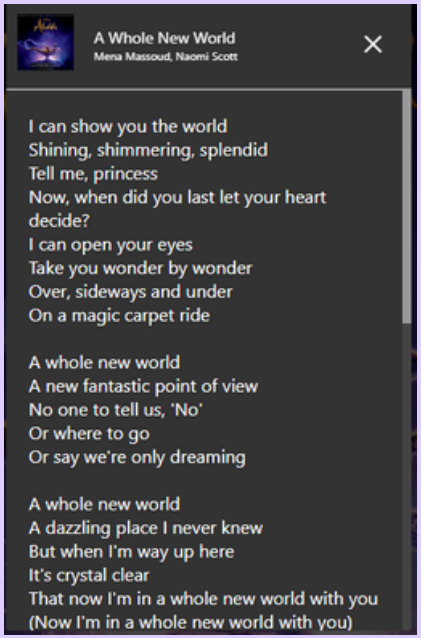
GO TO SITE!



음악이 재생중 일때 물결
모양으로 시퀄라이저 생성

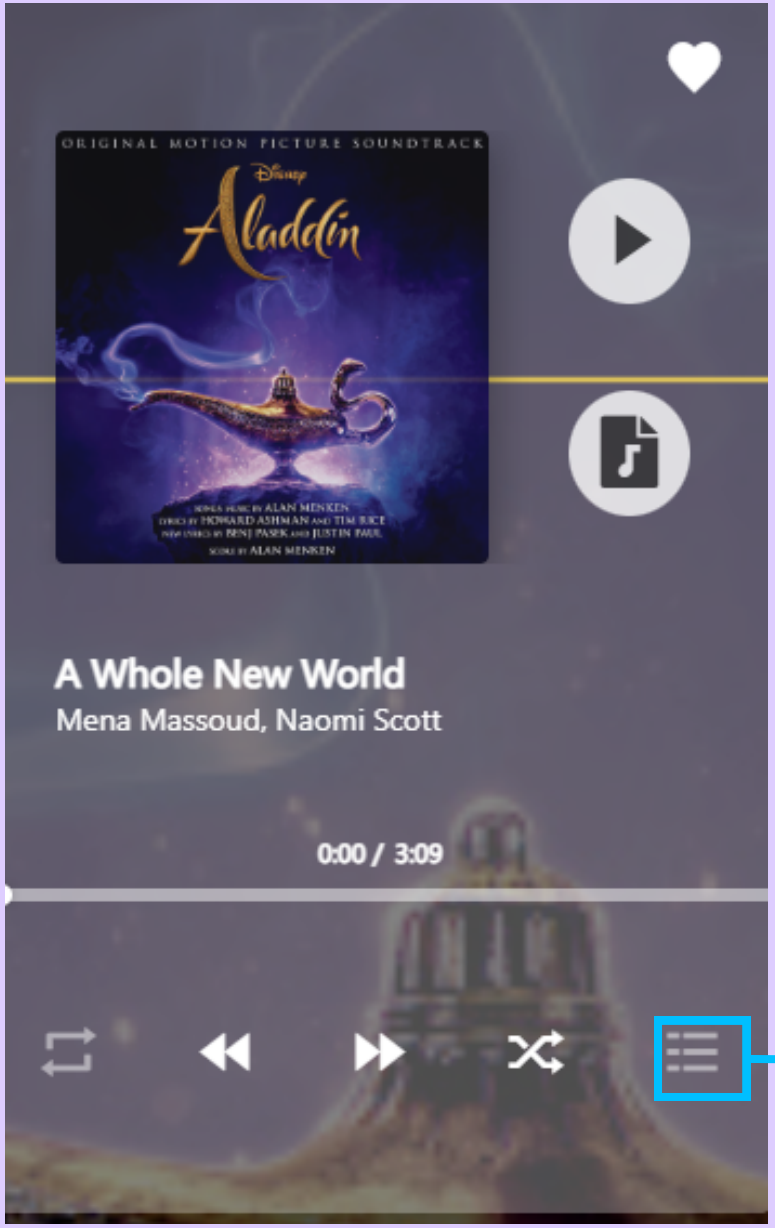


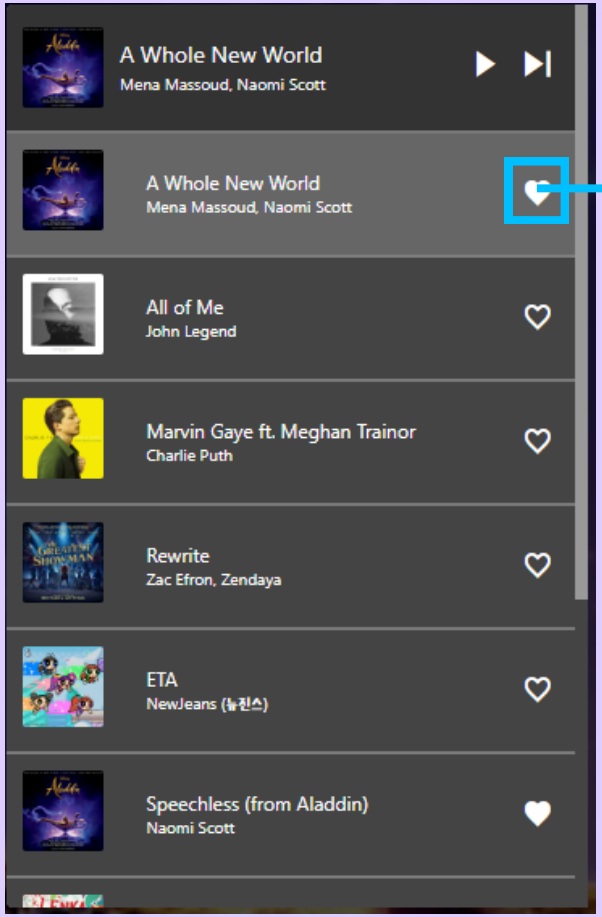
가사 출력 버튼



화면구성

GO TO SITE!





좋아요 된 음악 표시

재생리스트 생성 버튼

• FRONT-END

JavaScript / Deck, Music Class 설계

GO TO SITE!

```
class Music {
    constructor(id, musicData) {
        this.id = id;
        this.musicData = {
            ...musicData
        };
        this.likeCheck = false;
        this.playing = false;
    }
}
```

```
class Deck {
    constructor(id) {
        this._id = id;
        this._storage = [];
        this.length = 0;
    }
    get storage() {...}
    shiftItem() {...}
    unshiftItem(item) {...}
    popItem(){...}
    pushItem(item){...}
    getFirstItem(){...}
    getLength(){...}
}
```

자바스크립트에는 내장된 Deck 모듈이 없기 때문에, 이를 직접 설계하였습니다.
Deck에는 여러 Music 인스턴스들을 삽입하고, 이를 활용하여 뮤직 플레이리스트의 정방향 재생, 역순 재생, 랜덤 재생 기능을 구현하였습니다.

JavaScript / CustomMediaPlay, MusicVisualizer클래스 설계

GO TO SITE!

```
class CustomMediaPlay {
    constructor(id) {...}

    initLikeBtn() {...}
    nextContent() {...}
    prevContent() {...}
    shuffleContent() {...}
    listChange() {...}
    changeData() {...}
    listinit() {...}
    playBtnEvent() {...}
    pauseBtnEvent() {...}
    NextBtnEvent() {...}
    listBoxToMain() {...}
    lyricsBoxTOMain() {...}
    randomPlay() {...}

    init() {...}
    run() {...}
}
```

CustomMediaPlay 클래스에 시퀄라이저 기능을 제외한 모든 뮤직플레이어의 모든 기능을 구현하였습니다.

```
class MusicVisualizer {
    constructor() {...}

    animate(){...}

    init() {...}
    run() {...}
}
```

MusicVisualizer클래스에서 시퀄라이저 기능을 구현하였습니다.

JavaScript / 서버로 데이터 요청

GO TO SITE!

```
async function fetchData() {
    const response = await fetch(`${domain}init`);
    rawData = await response.json();
    const cmp = new CustomMediaPlay("cmp");
    rawData.forEach((value, index) => {
        cmp.rawDeck.pushItem(new Music(index + "_music", value));
    });
    cmp.run();
}
```

웹페이지가 호출되면 fetch를 이용하여 웹서버에게 요청을 보냅니다.
값을 JSON형식으로 반환받고 반환받은 값을 이용하여 Music 인스턴스를 생성합니다.

JavaScript / 서버로 데이터 요청

GO TO SITE!

```
async function fetchData() {
    const response = await fetch(`${domain}init`);
    rawData = await response.json();
    const cmp = new CustomMediaPlay("cmp");
    rawData.forEach((value, index) => {
        cmp.rawDeck.pushItem(new Music(index + "_music", value));
    });
    cmp.run();
}
```

웹페이지가 호출되면 fetch를 이용하여 웹서버에게 요청을 보냅니다.
값을 JSON형식으로 반환받고 반환받은 값을 이용하여 Music 인스턴스를 생성합니다.

JavaScript / 음악 재생 기능 구현

GO TO SITE!

```
nextContent() {
    this.rawDeck.pushItem(this.rawDeck.shiftItem());
    this.initLikeBtn();
    this.changeData("next");
}

prevContent() {
    this.rawDeck.unshiftItem(this.rawDeck.popItem());
    this.initLikeBtn();
    this.changeData("prev");
}

shuffleContent() {
    let prevId = this.rawDeck.storage[0].id;
    const randomNum = Math.floor(Math.random() * (this.rawDeck.storage.length - 1)) + 1;
    for (let i = 0; i < randomNum; i++) {
        this.rawDeck.unshiftItem(this.rawDeck.popItem());
    }
    this.changeData("shuffle", randomNum)
}
```

정방향 재생 이벤트 발생시 Deck의 첫번째 데이터를 shift와 push 메소드를 이용하여 맨 뒤로 이동시킵니다. 반대로 역순 재생이 발생했을때는 반대로 동작합니다.

랜덤재생 이벤트 발생시는 Deck에 있는 Music인스턴스 개수만큼 랜덤값을 생성하고 그 랜덤값 만큼 앞에서 했던 작업을 반복합니다.

JavaScript / 좋아요, 취소 버튼 업데이트 기능

GO TO SITE!

```
disLikeBtn.addEventListener('click', () => {
    disLikeBtn.classList.add('hidden');
    likeBtn.classList.remove('hidden');
    this.rawDeck.storage[0].musicData.checklike = "true"
    axios.get(`${domain}update`,
        {
            params: {
                type: 'like',
                id: this.rawDeck.storage[0].musicData.id
            }
        }).then(response => {
            console.log(response.data);
            this.listChange();
        })
        .catch(error => {
            console.error(error);
        });
});
```

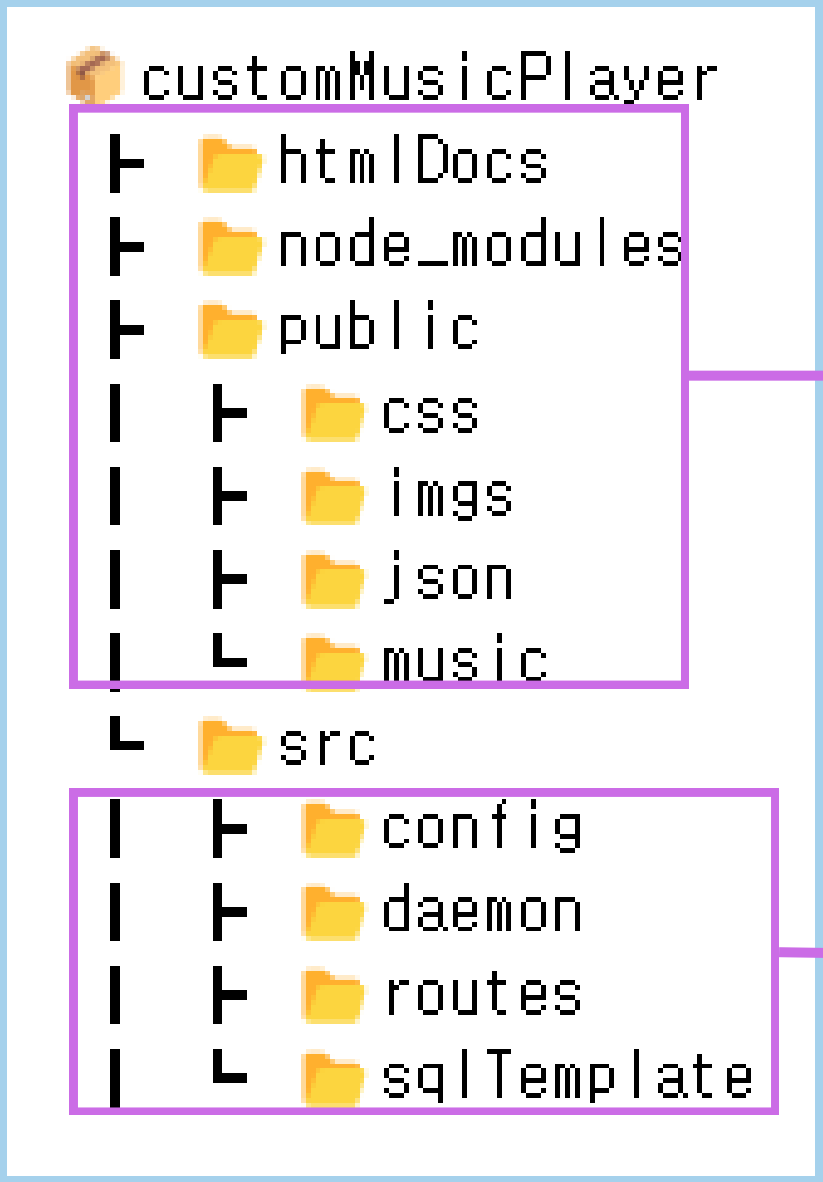
```
likeBtn.addEventListener('click', () => {
    disLikeBtn.classList.remove('hidden');
    likeBtn.classList.add('hidden');
    this.rawDeck.storage[0].musicData.checklike = "false"
    axios.get(`${domain}update`,
        {
            params: {
                type: 'dislike',
                id: this.rawDeck.storage[0].musicData.id
            }
        }).then(response => {
            console.log(response.data);
            this.listChange();
        })
        .catch(error => {
            console.error(error);
        });
});
```

좋아요, 싫어요 버튼을 클릭했을때 Axios 모듈을 사용하여 POST방식으로 웹서버에 업데이트 요청 후 요청이 정상적으로 반환 되면 데이터를 업데이트합니다.

• BACK-END

Node / 서버 폴더 구조

GO TO SITE!



사용자 인터페이스 담당 부분

환경 설정 , 실행,
라우팅 , SQL 폴더 구성

MySQL / 테이블 설계

GO TO SITE!

	#	이름	종류	데이터정렬방식	보기	Null	기본값	설명	추가
☐	1	**id** 🔑	int(11)			아니오	*없음*		AUTO_INCREMENT
☐	2	**singer**	varchar(100)	utf8mb4_general_ci		예	*NULL*		
☐	3	**title**	varchar(255)	utf8mb4_general_ci		예	*NULL*		
☐	4	**imgName**	varchar(255)	utf8mb4_general_ci		예	*NULL*		
☐	5	**fileName**	varchar(255)	utf8mb4_general_ci		예	*NULL*		
☐	6	**lyrics**	text	utf8mb4_general_ci		예			
☐	7	**checklike**	char(255)	utf8mb4_general_ci		예	*NULL*		

Music 인스턴스를 만들기 위한 데이터를 저장 하기 위해 설계한 테이블입니다.
곡에 대한 기본적인 정보와 좋아요를 체크 할 수 있는 속성들로 구성되어 있습니다.

Express / Config

GO TO SITE!

```
class Daemon {
    constructor(id) {
    ...
      this.serviceInfo = require("../config/service");
      this.dbInfo = require("../config/database");
      this.mysql = require("mysql");
      this.connection = this.mysql.createConnection(this.dbInfo);

    }
    daemonReady() {
     ...
          `Open ${this.serviceInfo.port} PORT !! \nhttp://localhost:${this.serviceInfo.port}
    }
}
```

```
const serverInfo = {
    port: ...,
    domain: 'http://localhost:.../'
}

module.exports = serverInfo;
```

```
module.exports = {
    host: "localhost",
    user: "...",
    password: "...",
    port:3306,
    database: "..."
}
```

Config 파일을 별도의 모듈로 분리하여 서비스 관련 환경 설정 정보를 관리하고, 실행 파일에서 require을 사용하여 이를 불러옵니다. 이를 통해 민감한 정보가 코드에 노출되지 않아 보안이 강화되며, 설정이 체계적으로 관리되어 추후 수정이 용이해집니다.

Express / Routes

GO TO SITE!

```
const path = require('path');
const sqlTemplate = require("../sqlTemplate

const routeRoot = (req, res) => {...}

const routeUpdate = (req, res) => {...}

const routeInit = (req, res) => {...}

module.exports = routeRoot;
module.exports = routeUpdate;
module.exports = routeInit;
```

```
runRoute() {
    this.app.get("/", (req, res) => {
      const routeRoot = require("../routes/routeRoot");
      routeRoot(req, res);
    });
    this.app.get("/update", (req, res) => {
      const routeUpdate = require("../routes/routeUpdate");
      routeUpdate(req, res);
    })
    this.app.get("/init", (req, res) => {
      const routeInit = require("../routes/routeInit");
      routeInit(req, res);
    }
    );
  }
```

Routes파일에 각 라우트(경로)로 접근 했을때 수행하는 작업을 모듈화 시킨 후 실행파일에서 각 모듈을 불러와서 실행합니다.

Express / SqlTemplate

GO TO SITE!

```
module.exports = {
    selectMusicAll : "select * from songs order by id desc;",
    updateTrue: "update songs set checklike = 'true' where id =",
    updateFalse: "update songs set checklike = 'false' where id =",
}
```

```
const sqlTemplate = require("../sqlTemplate/sqlcommand");

const routeUpdate = (req, res) => {
    switch (req.query.type) {
        case "like":
            const sql = `${sqlTemplate.updateTrue} ${req.query.id}`;
            ...
        case "dislike":
            const sql2 = `${sqlTemplate.updateFalse} ${req.query.id}`;
            ...
    }
    res.send("succeed update!");
}
```

Sql 구문 또한 하나의 모듈로 관리하여 추후 수정이 용이해지도록 구현하였습니다.

Express / daemon , index.js

GO TO SITE!

```
class Daemon {
    constructor(id) {
      this.id = id;
      this.express = require("express");
      this.app = this.express();
      this.path = require("path");
      this.axios = require("axios");
      this.serviceInfo = require("../config/service");
      // DB
      this.dbInfo = require("../config/database");
      this.mysql = require("mysql");
      this.connection = this.mysql.createConnection(this.dbInfo);
    }
    daemonReady() {...}
    runRoute() {...}
    run() {
      this.daemonReady();
      this.runRoute();
    }
  }

  const daemon = new Daemon("daemon1");
  module.exports = daemon;
```

```
const daemon = require( "./daemon/daemon" );
daemon.run();
```

Daemon 클래스를 설계하여 Express 웹 서버의 모든 기능을 수행하도록 구현하였으며, index.js에서는 단순히 Daemon 클래스를 호출하여 서비스를 실행하도록 구성하였습니다.

● 후기

후기

GO TO SITE!

웹 서버와 데이터베이스를 활용한 첫 번째 프로젝트

처음 웹 서버와 데이터베이스를 활용한 프로젝트를 설계할 때 막연한 두려움이 있었습니다. 그러나 웹 서버를 체계적으로 모듈화하여 구현하고, 데이터베이스와의 연동 작업을 진행하면서 백엔드 개발에 대한 자신감을 얻게 되었습니다. 특히, class 문법을 적극적으로 활용하며 객체지향 프로그래밍에 대한 이해를 더욱 깊이 할 수 있었던 의미 있는 프로젝트였습니다.

또한, 서버와 클라이언트 간의 통신에서 JSON 데이터 형식을 익히고, Axios 모듈과 자바스크립트 내장 함수 fetch를 활용한 비동기 통신 개념을 이해할 수 있어 유익한 경험이었습니다.

다만, 아쉬운 점은 백엔드 구현보다 프론트엔드 작업에 더 많은 시간을 소요했다는 것입니다. 특히, 코드보다는 시각적인 CSS 작업에서 많은 어려움을 겪었습니다. 이를 계기로 부족함을 깨닫고 CSS를 더욱 깊이 공부할 계획입니다. 또한, 데이터베이스 테이블을 설계할 때 너무 원시적인 값들만 고려했다는 점도 아쉬웠습니다. 이번 프로젝트에서는 많은 데이터를 다룰 필요가 없었지만, 더 세부적으로 고민하며 테이블을 설계했다면 더욱 효율적인 구조가 되었을 것이라는 생각이 들었습니다.
또한, 유저 권한 관리에 대한 고려 없이 root 계정으로 작업을 진행한 점도 반성하게 되었습니다. 앞으로는 특정 서비스에 맞는 유저 계정을 생성하고, 적절한 권한을 부여한 후 데이터베이스에 접근하는 연습을 해야겠다고 다짐했습니다.

이번 프로젝트를 통해 서버와 클라이언트 간의 데이터 처리 방식에 대해 배우고, 데이터베이스를 활용하여 값을 다루는 경험을 쌓을 수 있었습니다. 이를 바탕으로 앞으로 더 다양한 서비스를 개발할 수 있을 것이라는 기대감이 생긴 뜻깊은 프로젝트였습니다.

THANKS FOR WATCHING MY PRESENTATION

PERSONAL PROJECT

서동현