웹응용과제

Profiler 프로그램 분석

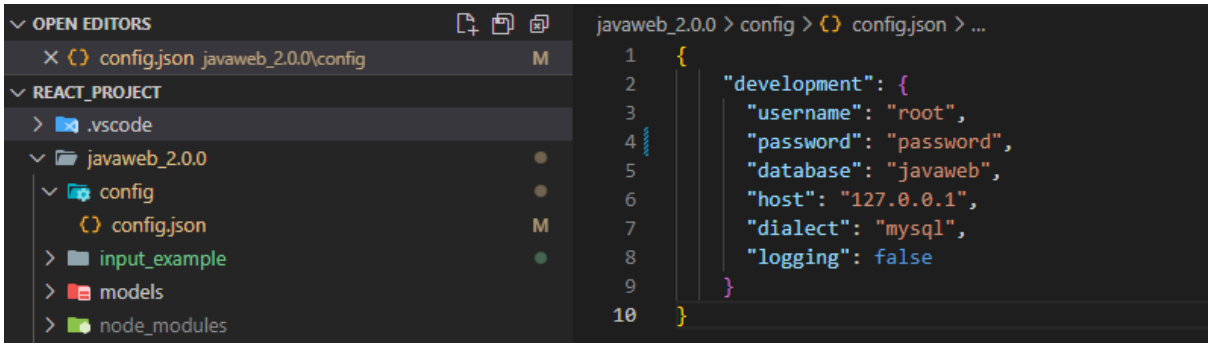
과 목	웹응용기술
담당 교수	강영명 교수
팀 원	20210854 오현영

<목차>

<그림 목차>

1. (제공된) 과거 수행한 소스 코드 분석

1.1 데이터베이스 연결 설정(config/config.json)



[그림 1] 데이터베이스 연결 설정

위 파일은 데이터베이스 연동을 위해 필요한 사용자의 비밀번호, 데이터베이스명, 호스트, DB 종류 등 중요한 설정 정보를 환경별로(ex. development) 미리 정의해두는 파일이다.

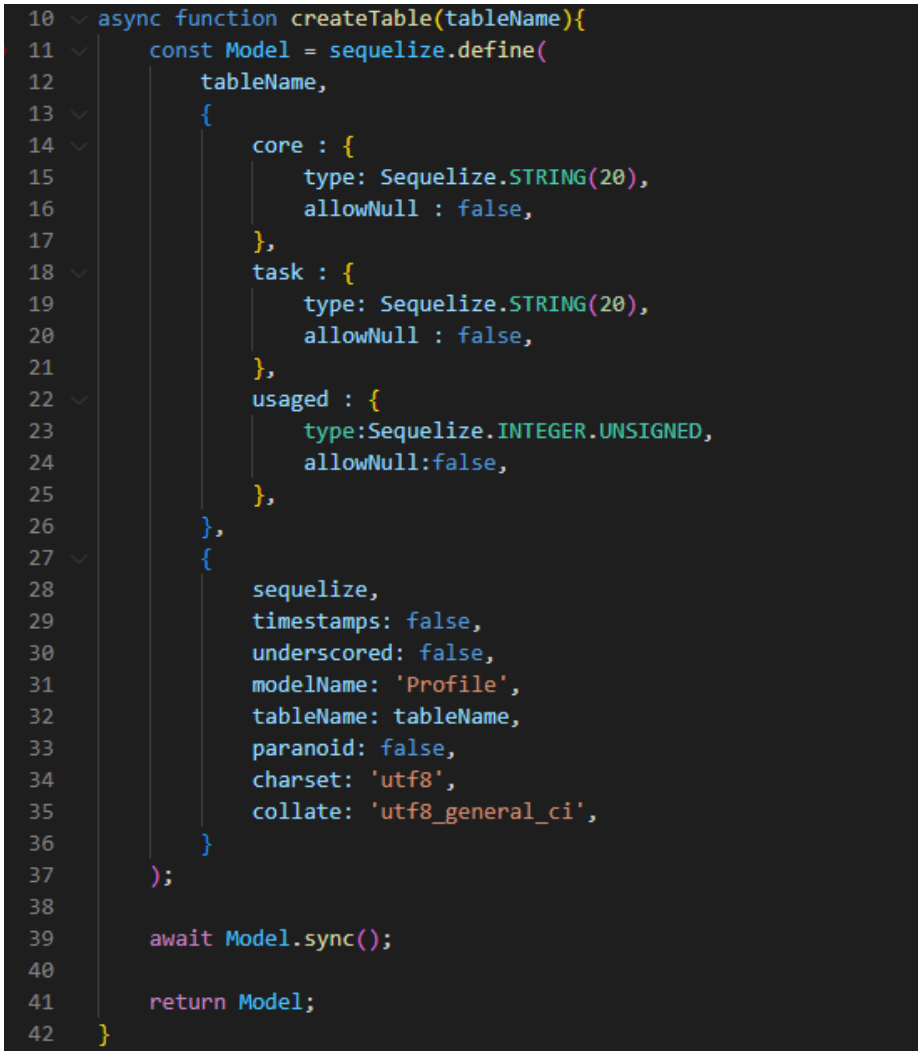
1.2 models

1.2.1 index.js

```
const Sequelize = require('sequelize');

const env = process.env.NODE_ENV || 'development';
const config = require('../config/config')[env];
const db = {};

const sequelize = new Sequelize(config.database, config.username, config.password, config);


```

[그림 2] index.js 모듈 불러오기

먼저 "sequelize"를 활용하기 위해 불러옵니다. 현재 작업이나 데이터베이스 환경 등을 미리 작성한 환경설정 파일과 "config" 파일의 정보를 통해 불러온다. 그 후에 불러온 설정 정보를 바탕으로 Sequelize 객체를 생성한다.



```
async function createTable(tableName){
    const Model = sequelize.define(
        tableName,
        {
            core : {
                type: Sequelize.STRING(20),
                allowNull : false,
            },
            task : {
                type: Sequelize.STRING(20),
                allowNull : false,
            },
            usaged : {
                type:Sequelize.INTEGER.UNSIGNED,
                allowNull:false,
            },
        },
        {
            sequelize,
            timestamps: false,
            underscored: false,
            modelName: 'Profile',
            tableName: tableName,
            paranoid: false,
            charset: 'utf8',
            collate: 'utf8_general_ci',
        }
    );

    await Model.sync();

    return Model;
}
```

[그림 3] 시퀄라이저 Model 정의

테이블 생성 함수("createTable")는 tableName 을 매개변수로 받아 데이터베이스에서 새로운 테이블을 동적으로 생성한다.

1. core; STRING(20)타입으로, allowNull: false 를 통해 Null 값을 허용하지 않는다.
2. task: STRING(20)타입으로, 마찬가지로 Null 값을 허용하지 않는다.
3. Usaged: INTEGER.UNSIGNED 타입으로, 마찬가지로 Null 값을 허용하지 않는다.

"await Model.sync()"를 통해 정의된 "Model"에 따라 실제 데이터베이스에 해당 테이블을 생성하거나, 이미 존재하는 경우 스키마를 동기화하여 비동기적으로 처리한다.

```
async function dropTable(tableName) {
    try {
        await sequelize.query(`DROP TABLE IF EXISTS \`${tableName}\``);
        console.log(`테이블 '${tableName}'이(가) 삭제되었습니다.`);
    } catch (error) {
        console.error(`테이블 삭제 중 오류가 발생했습니다: ${error}`);
    }
}

```

[그림 4] 테이블 삭제 함수

테이블 삭제 함수(dropTable)은 주어진 "tableName"에 해당하는 테이블을 데이터베이스에서 삭제하는 함수이다. 만약 해당 "tableName"의 테이블이 데이터베이스에 존재하는 경우 삭제를 하고, 그렇지 않은 경우엔 오류 메시지를 출력하도록 하였다.

```
async function createDynamicTable(profile){
    // console.log(profile);
    const tableName = profile[0][0];
    const DynamicModel = await createTable(tableName);

    let core_row = -1;
    for(let row = 1; row<profile.length; row++){
        if(core_row == -1){
            core_row = row;
            continue;
        }
        if(profile[row].length==1){
            core_row = -1;
            continue;
        }
        for(let column = 1; column < profile[row].length; column++){
            try{

            await DynamicModel.create({
                task: profile[core_row][column-1],
                core : profile[row][0],
                usaged : profile[row][column],
            });
            }catch(e){
                console.log(`Error: ${tableName} 파일 데이터 오류 발생`);
            }
        }
    }
}
```

[그림 5]동적 테이블 생성 함수

동적 테이블 생성 함수(createDynamicTable)는 "profile"이라는 2 차원 배열을 매개변수로 받아 동적으로 테이블을 생성하고 데이터를 삽입한다. 테이블 이름은 "profile"배열의 첫 번째 행, 첫 번째 열의 값으로 정의한다. 정의된 "tableName"을 사용하여 미리 정의한 'createTable'함수로 해당 테이블에 대한 Sequelize 모델을 얻는다. "core_row"는 데이터의 기준 행 인덱스이며, 배열의 두 번째 행부터 profile 배열의 길이만큼 반복을 진행한다. 이때 "core_row"의 값이 -1 이면, 즉 초기 상태이거나 이전 데이터 블록이 끝났을 경우, "core_row"변수에 현재 row 값을 업데이트하고 새로운 데이터 블록의 시작임을 알린다.

만약 해당 행의 profile 의 크기가 1 이라면, 다시 "core_row"변수를 초기화하여 새로운 행의 컬럼들을 받을 준비를 한다. 모두 해당하지 않는 경우 각 행의 컬럼들을 순회하며

각 데이터 포인트를 "DynamicModel"을 통해 데이터베이스에 새 레코드로 삽입한다.

```
85  async function getTableList() {
86      const query = 'SHOW TABLES'; // 데이터베이스별로 조회 방식이 다를 수 있으므로 사용하는 데이터베이스
87
88      // 쿼리 실행
89      const [results, metadata] = await sequelize.query(query);
90
91      // 테이블 목록 추출
92      const tableList = results.map((result) => result.Tables_in_javaweb); // 'database'는 실제
93
94      return tableList;
95  }
96
97
98  db.sequelize = sequelize;
99
100
101 module.exports = {db, createDynamicTable, sequelize, getTableList, dropTable};
102
```

[그림 6] 테이블 조회 함수

테이블 목록 조회 함수(getTableList)함수는 현재 데이터베이스에 존재하는 모든 테이블의 목록을 조회한다. "SHOW TABLES" 쿼리는 MySQL 데이터베이스에서의 표준 쿼리이므로 다른 데이터베이스에 연동하는 경우 수정해줄 필요가 있다. "sequelaize"인스턴스의 "query"메서드를 통해 SQL 쿼리를 실행하고 실제 테이블 이름만 추출하여 배열로 반환한다. 마지막으로 "db"객체에 "sequelize"인스턴스를 할당하여 외부에서로 데이터베이스 연결 객체에 접근할 수 있도록 하고, 다른 함수 또한 외부 파일에서 사용할 수 있도록 "exports'한다.

1.2.2 Profile.js

```
const Sequelize = require('sequelize');

class Profile extends Sequelize.Model{
    static initiate(sequelize,tableName){
        Profile.init({
            core : {
                type: Sequelize.STRING(20),
                allowNull : false,
            },
            task : {
                type: Sequelize.STRING(20),
                allowNull : false,
            },
            usaged : {
                type:Sequelize.INTEGER.UNSIGNED,
                allowNull:false,
            },
        },
        {
            sequelize,
            timestamps: false,
            underscored: false,
            modelName: 'Profile',
            tableName: tableName,
            paranoid: false,
            charset: 'utf8',
            collate: 'utf8_general_ci',
        });
    }

    static associations(db){

    }
};

module.exports = Profile;
```

[그림 7] Profile 테이블

이 코드는 Sequelize ORM 을 사용하여 데이터베이스 테이블의 구조를 정의한다. "Sequelize.Model" 클래스를 상속받아 데이터베이스와 매핑된 새로운 "Profile"클래스를 생성한다. 정적 메서드는 "Profile"모델을 초기화하고 모델을 데이터베이스와 연결하고 스키마를 정의한다. "associations"정적 메서드는 다른 모델과의 관계를 정의하는 데 사용한다. 현재 코드에서는 사용되지 않는다.

1.3. public

1.3.1 sequelize.js

```
let fileName=""; // 파일 이름 저장
let selete=""; // core or task

// 해당하는 id를 가진 테이블 내 모든 <tr>의 첫 번째 요소를 profileList에 저장
const profileList = document.querySelectorAll('#profile_list tr td:first-child');

//ProfileList의 각 항목 처리
profileList.forEach((el) => {
    //click에 대한 이벤트리스너 추가
    el.addEventListener('click', function () {
        // fileName에 profileList의 textContext 부분을 저장
        fileName = el.textContent;
        profileList.forEach((otherEl) => {
            otherEl.style.setProperty("background-color", "white");
        });
        this.style.setProperty("background-color", "#888888"); // 현재 선택된 항목에 대한 표현
        select = undefined; // 새로운 항목이 선택되었으므로 select 초기화
        // 만약 차트가 존재했다면 삭제제
        if (chart) {
            chart.destroy();
        }
        getdata(); // 선택된 fileName에 해당하는 데이터를 백엔드에서 불러옴
    });
});
```

[그림 8] ProfileList 이벤트 리스너

먼저 "profile_list" id 를 가진 테이블 내 모든 행의 가장 첫 번째 <td>요소를 선택하여 "profileList"에 저장한다. 각 항목을 처리하며 사용자가 목록의 항목을 클릭하는 경우 그 항목의 텍스트 콘텐츠를 'fineName'으로 저장한다. 모든 프로필 목록 항목의 배경색을 흰색으로 설정하고 선택된 항목에 대해서만 회색으로 표현하여 선택된 항목을 표현하였다. "select'는 "core"나 "task" 선택 변수인데 현재는 새로운 항목을 선택했으므로 초기화하고 만약 이미 차트가 존재하는 상황이었다면 차트를 제거하고 선택된 파일명에 해당하는 데이터를 백엔드로부터 불러온다.

```
// 아이디가 profile_form인 태그의 submit 이벤트에 대한 비동기 함수 정의
document.getElementById('profile_form').addEventListener('submit', async (e) => {
    e.preventDefault(); // 새로고침 방지

    // #input_profile인 선택자의 file들을 files에 저장
    const files = document.querySelector('#input_profile').files;
    let profiles = [];
    let is_error = false;
    if(!files){
        return alert('파일을 등록하세요');
    }
    const filePromises = Array.from(files).map((file) => {
        // 만약 파일이 txt 파일이라면
        if (file.name.split(".").pop().toLowerCase() === 'txt') {
            return new Promise((resolve, reject) => {
                // 해당 파일의 내용을을 읽고 파싱된 데이터를 profiles 배열에 저장한 후 Promise resolve
                readTextFile(file, (data) => {
                    profiles.push(data);
                    resolve();
                });
            });
        } else {
            alert(".txt파일만 입력해주세요");
            is_error = true;
        }
    });
```

[그림 9] txt 파일 처리 로직

Id 가 "profile_form"인 태그에 submit 이벤트에 대한 비동기 함수이다. Id 가 "Input_profile"인 태그의 파일들을 "files"변수에 저장한다. files 변수가 존재하지 않는다면, 즉 파일이 없다면 메시지를 출력한다. 파일명을 전처리하여 확장자를 추출한다 만약 "txt"라면 프로미스를 사용하여 해당 파일의 내용을 읽고 미리 정의해둔 "profiles" 배열에 업데이트 한 후에 프로미스를 해결한다.

```
    await Promise.all(filePromises); // 모든 파일의 읽기 작업이 종료될 때까지 대기

    if(!is_error){
        const response = await fetch('/profiles',{
            method: 'POST',
            headers: {
                'Content-Type' : 'application/json'
            },
            body: JSON.stringify(profiles) // profiles을 json 형태로 직렬화
        });

        if (response.ok) {
            response.json().then(data => {
                getList();
                alert(data.message);
            });
        } else {
            console.error('파일 전송 중 오류가 발생하였습니다.');
        }
    }
});
```

[그림 10] 파일 error 처리

파일을 읽는 과정에서 오류가 없었다면 "profiles"를 json으로 직렬화하여 백엔드의 "/profiles"로 POST요청을 보낸다. 응답 코드가 성공인 경우, JSON 메시지를 alert 창으로 표시하고 "getList()"함수를 호출하여 프로필 목록을 갱신한다. 그렇지 않은 경우 콘솔에 오류 메시지를 출력한다.

```
async function getList(){
    const res = await axios.get('profiles'); // axios를 통해 profiles에 get 요청
    const profiles = res.data; // 응답의 데이터를 profiles에 저장

    const tbody = document.querySelector('#profile_list tbody');
    tbody.innerHTML = ''; // 빈 문자열로 초기화 -> 기존 목록 초기화

    // 각 프로파일에 대해 동적 생성 및 이름 표시
    profiles.map(function(profile){
        // 테이블 행렬 생성
        const row = document.createElement('tr');
        const td = document.createElement('td');
        td.textContent = profile;
        td.className= 'text-center fw-semibold';
        td.addEventListener('click',function(){
            fileName = profile;
            const profileList = document.querySelectorAll('#profile_list tr td:first-child');
            profileList.forEach((otherEl) => {
                otherEl.style.setProperty("background-color", "white");
            });
            this.style.setProperty("background-color", "#888888");
            if (chart) {
                chart.destroy();
            }
            getdata();
        });
        // 현재 선택된 profile이 있다면, 배경색 유지
        if(profile==fileName) td.style.setProperty("background-color", "#888888");
        row.appendChild(td);
        const td2 = document.createElement('td');
        const btndrop = document.createElement('button');
        btndrop.textContent="삭제";
        btndrop.className="btn btn-danger";
        btndrop.addEventListener('click',function(){ deleteTable(`${profile}`); });
        td2.appendChild(btndrop);
        row.appendChild(td2);

        tbody.appendChild(row);
    });
}
```

[그림 11] getList() 함수 코드

```
116 async function deleteTable(name){
117     // 매개변수 name에 해당하는 profile 테이블 삭제 요청
118     await axios.delete(`profiles/drop/${name}`);
119     if(fileName==name && chart) {
120         chart.destroy();
121         const task_div = document.querySelector('#core');
122         task_div.innerHTML=""; //html의 'core'부분을 다시 초기화
123         const core_div = document.querySelector('#task');
124         core_div.innerHTML = ''; //html의 'task'부분을 다시 초기화
125         fileName = "";
126     };
127     setTimeout(getList,50);
128 }
```

[그림 12] deleteTable() 함수 코드

"getList" 함수는 백엔드에서 저장된 테이블 목록을 비동기적으로 불러와 웹 페이지에 렌더링하는 함수이다. Axios 라이브러리를 사용하여 profiles 앤드포인트에 GET 을 요청하고 그에 대한 응답 값을 res 에 저장한다. "profiles" 변수에 응답의 데이터를 저장한다. 만족하는 Id 의 <tbody>를 tbody 에 저장한 후에 빈 문자열로 초기화한다. 그런 다음 각 프로파일에 대해 <tr>과 <td>를 동적으로 생성하고, 이름을 표시한다. 항목 선택 표시는 위에서 정의했던 이벤트 리스너를 적용하였으며, 현재 선택된 것이 있다면 배경색을 유지한다. 여기서 삭제 버튼을 하나 추가하여 사용자가 해당 버튼을 클릭한 경우, "deleteTable()" 함수를 호출하여 해당 테이블을 삭제하도록 한다.

```
async function getdata(){

    const res = await axios.get(`profiles/data/${fileName}`);

    // 해당 파일의 cores, tasks 목록 데이터를 받음
    const cores = res.data.cores;
    const tasks = res.data.tasks;


    const task_div = document.querySelector('#core');
    task_div.innerHTML = 'select core : ';
    tasks.map(function(task){
        let button = document.createElement('button');
        button.className = 'btn btn-info me-2';
        button.textContent = task.core;
        button.addEventListener('click', function(){
            updateChart('task', task.core); // 버튼 클릭시 task.core에 맞는 chart 생성
            const coreDiv = document.getElementById('core');
            const coreBtns = coreDiv.getElementsByClassName('btn');
            for (let i = 0; i < coreBtns.length; i++) {
                coreBtns[i].className = "btn btn-info me-2";
            }
            const taskDiv = document.getElementById('task');
            const taskBtns = taskDiv.getElementsByClassName('btn');
            for (let i = 0; i < taskBtns.length; i++) {
                taskBtns[i].className = "btn btn-success me-2";
            }
            this.className = "btn btn-secondary me-2";
        });
        task_div.appendChild(button);
    });

    const core_div = document.querySelector('#task');
    core_div.innerHTML = 'select task : ';
```

[그림 13] getdata() 함수

현재 선택된 fileName 에 해당하는 데이터를 백엔드에서 불러와 "core"및 "task" 선택 버튼을 렌더링하는 비동기 함수이다. "core" 버튼 렌더링의 경우 "#core" id 를 가진 요소의 내용을 초기화하고 클래스 이름을 정의한다. 버튼 클릭시 task.core 에 대한 차트를 업데이트하고 선택 상태를 시각적으로 표현한다. 아래 "task"에 대한 로직도 동일하다.

```
function readTextFile(file, save) {
    const reader = new FileReader(); // 비동기적으로 읽음

    // 파일을 성공적으로 읽은 경우
    reader.onload = async function(event) {
        const contents = event.target.result; // 읽은 파일의 텍스트 내용 반환
        let line_parse = contents.split("\n");
        const parse = [[file.name]]; // 파싱된 데이터를 저장할 2차원 배열을 file.name을 첫번재 요소로 초기화
        for(let i=0; i<line_parse.length; i++){
            parse.push(line_parse[i].trim().split(/\t| |,|\//));
        }
        save(parse);
    };

    reader.onerror = function(event){
        console.error("잘못된 파일");
    }

    reader.readAsText(file, 'UTF-8');

}
```

[그림 14] readTextFile() 함수

이는 텍스트 파일을 읽는 함수이다. "FileReader" 객체를 생성하여 파일 내용을 비동기적으로 읽는다. 파일을 성공적으로 읽은 경우, 읽은 파일의 텍스트 내용을 "contents'에 저장하고 줄바꿈을 기준으로 분리한다. 파싱된 데이터를 2 차원 배열에 저장을 하기 위해 파일의 이름을 첫 요소로 하여 초기화한다. 반복을 통해 데이터를 파싱하고 파싱된 2 차원 배열 데이터를 저장한다. 파일 읽기 중 오류 발생 시 콘솔에 에러 메시지를 출력한다.

```
// chart.js를 사용하여 데이터를 시각화하는 핵심 함수
async function updateChart(type, choose_name){

    const profiler = document.getElementById('profiler').getContext('2d');
    if (chart) {
        chart.destroy();
    }

    // 데이터 로드, core인 경우
    if(type == 'core'){
        select = choose_name;
        const res = await axios.get(`profiles/taskdata/${fileName}/${select}`);
        const datas = res.data;

        labels = [];
        minData = [];
        maxData = [];
        avgData = [];
        // 백엔드에서 데이터를 불러와서 배열에 저장
        datas.forEach((data) => {
            labels.push(data.core);
            minData.push(data.min_usaged);
            maxData.push(data.max_usaged);
            avgData.push(data.avg_usaged);
        });

```

[그림 15] updateChart() 함수

이 함수는 차트 업데이트 함수이다. 우선 차트가 그려질 <canvas> 요소의 2D 렌더링 컨텍스트를 불러온다. 이미 차트가 존재한다면 해당 차트를 제거한다. 차트의 타입이 "core"인 경우 백엔드에서 task별 레이블, 최솟값, 최댓값, 평균값 데이터를 불러와서 각각의 배열에 저장한다. "task"인 경우에도 동일하게 작동하며, 차트를 그리기 위한 필수 정보가 없다면 함수를 종료한다.

```
283     chart = new Chart(profiler, {
284         type: `${chart_type}`, // 사용자가 선택한 차트 type
285         data: {
286           labels: labels,
287           datasets: [{
288             label: 'Min',
289             data: minData,
290             borderColor: 'rgba(0, 0, 255, 0.5)',
291             backgroundColor: 'rgba(0, 0, 255, 0.5)',
292           }, {
293             label: 'Max',
294             data: maxData,
295             borderColor: 'rgba(255, 0, 0, 1)',
296             backgroundColor: 'rgba(255, 0, 0, 0.5)',
297           }, {
298             label: 'Avg',
299             data: avgData,
300             borderColor: 'rgba(100, 255, 30, 1)',
301             backgroundColor: 'rgba(100, 255, 30, 0.5)',
302           }]
303         },
304         // 차트의 시각적인 설정 정의
305         options: {
306           maintainAspectRatio: false,
307           scales: {
308             y: {
309               beginAtZero: true
310             }
311           },
312           plugins: {
313             title: {
314               display: true,
315               text: `${fileName}의 ${select} 정보`,
316               font: {
317                 size: 30
318               }
319             }
320           }
321         },
322     });
323 
324 }
```

[그림 16] 차트 생성

마지막으로 차트를 생성하며 사용자가 선택한 타입으로 차트를 그린다. 각 데이터를 활용하여 차트 데이터를 구성하고, 각 데이터는 고유한 색상과 레이블을 가진다.

1.4 Routes

1.4.1 index.js

```
const express = require('express');
const router = express.Router();

 //db, createDynamicTable, sequelize, getTableList, dropTable 가져옴
const { getTableList } = require('../models/index');
//이중에서 DB 불러오기 기능만 사용

//이벤트 핸들러(테이블을 가져온다, 오류 발생시 에러 메시지 출력)
router.get('/', async (req,res)=>{
    //DB 불러오기 실행
    getTableList()
        //정상적 수행
        .then((tableList) => {
            //console.log('테이블 리스트:', tableList);
            res.render('index', {tableList});
        })

        //오류메시지
        .catch((error) => {
        console.error('테이블 리스트 조회 중 오류가 발생하였습니다:', error);
        });
});

module.exports = router;     //router(DB불러오기)기능 배포
```

[그림 17] index.js

이 코드는 Express 라우터의 한 부분으로 Routes/index.js 에 앞서 정의한 데이터베이스 조회 함수인 "getTableList()"를 불러온다. Express 의 Router 객체를 생성하여 특정 경로에 대한 요청을 처리하는 미들웨어와 라우트 핸들러를 정의하는데 사용한다. 루트 경로로 GET 요청이 들어왔을 때 Promise 가 정상적으로 해결되면 테이블리스트를 이용하여 웹페이지를 렌더링하고 응답을 한다. 오류가 발생하는 경우 콘솔에 오류 메시지를 출력한다.

1.4.2 profile.js

```
const express = require('express');    //express 서버 구성
const router = express.Router();    //라우팅을 위한 객체 생성
const { createDynamicTable, getTableList, sequelize, dropTable } = require('../models/index');
const profile_model = require('../models/profile');
```

[그림 18] profile.js

"profile"에 대한 미들웨어와 라우트 핸들러 정의하는 코드이다. "/models/index"에서 정의한 여러가지 DB 함수와 sequelize 를 불러온다.

```
// 메인 페이지에서, POST(생성)요청시 처리(input 파일 파싱하는 부분)
router.post('/', async (req, res) => {
    const profiles = req.body;
    //클라이언트요청문으로 보내는 body정보 가져옴(input 파일 내용 가져오는거, 어느정도 정제된 input 파일, 3차원 배열 형태임)
    let count = 0;
    //console.log(profiles, '프로파일 입니다~~~~~~~~~~');
    try {
        const tableList = await getTableList();    //DB 테이블 불러오기 기능

    for (let file_num = 0; file_num < profiles.length; file_num++) {
        //소문자로 core,task명 전부 바꾸고, 파일 확장자 제거
        profiles[file_num][0][0] = profiles[file_num][0][0].toLowerCase().slice(0,-4);

        // 이미 존재하는 Table이면 넘기는 처리를 하는 부분
        if (tableList.includes(profiles[file_num][0][0])) {
          console.log("이미 존재하는 파일입니다");
          continue;
        }

        await createDynamicTable(profiles[file_num]);    //동적 테이블을 생성(Core,Task의 갯수에 따라서)
        count++;
    }
        //count는 몇개의 input 파일을 처리했는지를 의미한다.(input 파일 여러개 처리 가능함)
        if(count===0){
            res.json({ status: 'success', message: `저장 가능한 파일이 존재하지 않습니다.` });
        }else if(count==profiles.length){
            res.json({ status: 'success', message: `${count}개의 프로파일이 정상적으로 저장되었습니다.` });
        }else{
            res.json({ status: 'success', message: `중복된 이름의 파일을 제외한 ${count}개의 프로파일이 저장되었습니다.
        }


        // 오류 발생시 처리
    } catch (error) {
        console.error('오류가 발생하였습니다:', error);
        res.json({ status: 'error', message: '오류가 발생하였습니다.' });
    }
});
```

[그림 19] profiles.js 시퀄라이저 로직

루트 경로에서 POST 요청 시 응답의 데이터 부분을 profiles 에 저장한다.
데이터베이스에 존재하는 모든 테이블의 이름 목록을 비동기적으로 가져오는데, 이는 새로 업로드 될 파일 이름과의 중복 여부를 확인하는데 사용된다. 파일 이름을 가져온 후 정제한다. 그런 다음 중복 테이블을 확인해서 이미 존재하는 경우 콘솔에 메시지를 출력한다. 그렇지 않은 경우 core, task 의 개수에 따라 동적 테이블을 생성한다. Count 의 값에 따라 메시지를 JSON 형태로 응답한다.

```
// DB에서 table 목록 전체를 불러오고, Json 문서 형식으로 변환해서 응답하는 부분
router.get('/', async (req,res)=>{
    const tableList = await getTableList();
    res.json(tableList);
});

// 해당 테이블 명을 가진 Table을 호출하는 부분이다.(해당 inputfile을 클릭시, 불러오는 부분)
router.get('/data/:tableName', async (req,res)=>{
    try{
        const {tableName} = req.params;

        const tableList = await getTableList();    //1개의 테이블을 조회

        // 해당 table이 db에 존재하지 않으면, 오류 처리
        if(!tableList.includes(tableName)){
            return res.status(404).json({error:'존재하지 않는 파일입니다.'});
        }

        // 해당 table 모델을 초기화한다.
        profile_model.initiate(sequelize, tableName);

        // 테이블의 모든 데이터를 가져와서, datas에 저장함
        const datas = await profile_model.findAll();

        // task 기준 core처리 현황을 불러옴(
        const tasks = await profile_model.findAll({
            attributes: [sequelize.fn('DISTINCT', sequelize.col('core')), 'core'],
        });

        // core 기준 task처리 현황을 불러옴
        const cores = await profile_model.findAll({
            attributes: [sequelize.fn('DISTINCT', sequelize.col('task')), 'task'],
        });

        // json 문서 형태로 응답(모든 데이터, core기준 task데이터, task기준 core데이터)
        res.json({datas: datas, cores : cores, tasks : tasks});
    }catch(error){    // 오류 발생시
        console.error('데이터 조회 오류', error);
    }
});
```

[그림 20] 테이블 라우터 정의

테이블 이름에 대한 GET 라우터 정의이다. 응답의 파라미터 값을 추출하여 "tableName"에 저장한다. 그런 현재 데이터베이스에 존재하는 모든 테이블의 이름 목록을 비동기적으로 가져와서 해당 테이블이 데이터베이스에 존재하지 않으면 오류로 처리한다. 다음으로 Profile.js 에서 정의된 Profile 모델 클래스의 정적 메서드를 호출하여 해당 테이블에 대한 모델을 초기화한다. 테이블의 모든 데이터를 가져와서 "data"변수에 저장하고 'core"컬럼의 고유한 값들을 조회하여 tasks 변수에 저장한다. 마찬가지로 "task"컬럼의 고유한 값들을 조회하여 core 변수에 저장한다. 모든 조회된 데이터들은 JSON 문서 형태로 응답한다.

1.5 models

1.5.1 app.js

app.js 는 express 모듈을 불러와 웹 서버 인스턴스를 초기화하고 포트 번호를 설정한다. 웹을 동적으로 생성하기 위해 Nunjucks 템플릿 엔진을 설정하고 렌더링할 수 있도록 뷰 엔진을 지정한다. Models/index.js 에서 불러온 sequelize 인스턴스를 통해 데이터베이스 동기화 등의 작업을 한다. App.use()를 사용하여 다양한 미들웨어를 장착한다. 해당 app.js 에서는 정적 파일 제공, 본문 파싱 등이 포함된다. 라우터를 연결하여 각 경로에 대한 요청을 처리한다.

2. 후기

해당 프로젝트를 분석하면서 많은 어려움이 있었다. 강의에서 배운 내용 범위 내에서 진행된 프로젝트이지만 전체적인 프로그램을 개발하는데 어려움, 두려움이 있었지만 이번 과제를 통해 전체적인 백엔드, 프론트엔드 간의 유기적인 연동 과정, 데이터베이스 연동, 그리고 프로그램이 개발되는 전반적인 흐름에 대해 정확히 이해할 수 있었다. 막연하게만 느껴졌던 프로그램을 개발 과정에서 각 기능에 대한 파일의 구조, 서버, 데이터베이스 연동, 데이터 처리 등에 대한 확신을 가질 수 있게 된 점이 특히 의미 있었다. 이러한 경험을 바탕으로, 앞으로 더욱 크고 복잡한 프로그램을 구현하기 위한 심층적인 학습을 지속할 계획이다.

3. github 확장 프로그램

https://github.com/hyunyoungDA/nodejs_lab

해당 링크로 접속 시 Profiler 확장 프로그램을 확인할 수 있다.