# Front-End & Back-End

# 웹 서버 개발 명세서

문서번호 : DS-104

VER1.0

server > JS server.js > ...

```
import express from 'express';
import cors from 'cors';
import dotenv from 'dotenv';
import sendMail from './sendMail.js';
import { scanUpload, scanUploadFile } from './uploadFileScan.js';
import { packUpload, packUploadFile } from './uploadFilePack.js';
import { downloadEncryptedFile } from './downloadFile.js';
// 환경 변수 설정을 위한 dotenv.config() 호출
dotenv.config();

const app = express();

// CORS 옵션을 적용
app.use(cors());

// JSON 형식의 데이터 처리
app.use(express.json());

// 이메일 전송을 위한 POST 요청
app.post('/send-email', sendMail);

// 파일 업로드를 위한 POST 요청(VirusScan)
app.post('/upload-file-scan', scanUpload.single('file'), scanUploadFile);

// 파일 업로드를 위한 POST 요청(Packing)
app.post('/upload-file-pack', packUpload.single('file'), packUploadFile);

// 파일 다운로드를 위한 GET 요청(Packing)
app.get('/download-file/:fileId', (req, res) => {downloadEncryptedFile(req, res);});

// 서버 실행
app.listen(5000, () => {
    console.log('서버가 5000번 포트에서 실행 중입니다.');
});
```

| | |
|---|---|
| 목적 | server.js 파일은 Express.js를 사용하여 다양한 서비스 요청을 처리하는 백엔드 서버이다. 서버는 CORS 정책을 허용하고 JSON 형식의 데이터를 처리하며, 5000번 포트에서 실행됨. |
| 파일명 | server.js |

```javascript
import nodemailer from 'nodemailer';

async function sendMail(req, res) {
    // 클라이언트로부터 전달된 요청(body)에서 name, email, message 데이터를 추출
    const { name, email, message } = req.body;

    // Nodemailer 설정 - 네이버
    let transporter = nodemailer.createTransport({
        host: 'smtp.naver.com', // 네이버 SMTP 서버
        port: 587, // 네이버 SMTP 포트
        secure: false, // true는 SSL사용 false는 TLS사용
        auth: {
            user: process.env.EMAIL_USER, // 네이버 이메일 계정
            pass: process.env.EMAIL_PASS, // 네이버 이메일 비밀번호
        },
    });

    // 이메일 옵션 설정
    let mailOptions = {
        from: process.env.EMAIL_USER, // 네이버 SMTP 서버 제한때문에 발신자, 수신자 동일하게 함.
        to: process.env.EMAIL_USER, // 네이버 이메일로 수신
        subject: `문의사항: ${email}`, // 제목
        text: `${name} 님의 문의 내용:\n\n${message}\n\n\n\n\n\n\n발신자 이메일: ${email}`,
        replyTo: email, // 답장 시 사용할 이메일
    };

    // try-catch를 사용해 이메일 전송 시도
    try {
        // 이메일을 async(비동기)로 전송하고, 전송 완료 시까지 기다림
        await transporter.sendMail(mailOptions);
        // 이메일이 성공적으로 전송되면 클라이언트에게 성공 응답을 보냄
        res.json({ success: true });
    } catch (error) {
        // 이메일 전송에 실패하면 오류 로그를 출력하고 실패 응답을 보냄
        console.error('이메일 전송 오류:', error);
        res.json({ success: false });
    }
}

export default sendMail;
```

| | |
|---|---|
| 목적 | nodemailer를 사용하여 네이버 SMTP 서버와 통신하며, 사용자가 입력한 이름, 이메일, 문의 내용을 발신자의 이메일로 전송. |
| 파일명 | sendMail.js |

```
import { MongoClient, GridFSBucket } from 'mongodb'; // MongoDB 클라이언트와 GridFSBucket 모듈을 가져옴
import crypto from 'crypto'; // 파일의 MD5 해시를 계산하기 위해 crypto 모듈을 가져옴
import multer from 'multer'; // 파일 업로드를 처리하기 위한 multer 모듈을 가져옴
import dotenv from 'dotenv'; // 환경 변수를 로드하기 위한 dotenv 모듈을 가져옴

dotenv.config(); // .env 파일에 저장된 환경 변수를 로드

// MongoDB 연결 정보 (환경 변수로부터 MongoDB URI를 로드)
const MONGO_URI = process.env.MONGO_URI; // MongoDB URI를 환경 변수에서 가져옴
const DB_NAME = 'vsapi'; // MongoDB에서 사용할 데이터베이스 이름
const FILES_COLLECTION = 'file'; // 파일 메타데이터가 저장될 컬렉션 이름

let db; // MongoDB 데이터베이스 연결을 위한 변수

// Multer 설정: 메모리에서 파일을 처리하는 미들웨어
const scanUpload = multer(); // Multer는 파일을 메모리 내에서 처리함

// MongoDB 연결 함수: 서버 시작 시 데이터베이스 연결을 설정하는 함수
const connectToMongoDB = async () => {
    try {
        // MongoDB에 연결 (MONGO_URI를 사용)
        const client = await MongoClient.connect(MONGO_URI);
        db = client.db(DB_NAME); // 데이터베이스 객체를 설정
        console.log('uploadFileScan.js Connected to MongoDB'); // 연결 성공 메시지 출력
    } catch (error) {
        // 연결 실패 시 오류 메시지 출력
        console.error('MongoDB Connection Error:', error);
    }
};

// MD5 해시 계산 함수: 파일 데이터에 대해 MD5 해시값을 계산
const calculateMd5 = (fileBuffer) => {
    return crypto.createHash('md5').update(fileBuffer).digest('hex');
};

```

```
// Signature ID 생성 함수: 고유한 signature_id를 생성 (오늘 날짜 기반)
const generateSignatureId = async () => {
    const today = new Date().toISOString().slice(0, 10).replace(/-/g, '');
    const collection = db.collection(FILES_COLLECTION); // 'file' 컬렉션을 선택

    // 오늘 날짜로 시작하는 signature_id의 개수를 세서 다음 번호를 생성
    const count = await collection.countDocuments({
        signature_id: { $regex: `^${today}-` }, // 오늘 날짜로 시작하는 signature_id를 찾음
    });
    const nextId = count + 1; // 다음 ID 번호를 결정
    return `${today}-${nextId.toString().padStart(3, '0')}`;
};

// 지연 함수
const delay = (ms) => new Promise(resolve => setTimeout(resolve, ms));

// 파일 업로드를 처리하는 함수: 사용자가 업로드한 파일을 MongoDB GridFS에 저장하는 함수
const scanUploadFile = async (req, res) => {
    try {
        const fileBuffer = req.file.buffer; // multer를 통해 업로드된 파일의 데이터를 메모리에서 가져옴
        const filename = req.file.originalname; // 업로드된 파일의 원본 파일명을 가져옴
        const uploadIp = req.ip; // 파일을 업로드한 사용자의 IP 주소를 가져옴

        // 파일의 MD5 해시값을 계산하여 무결성을 확인
        const filehash = calculateMd5(fileBuffer);

        let signature_id;

        // GridFS에 파일을 저장: MongoDB의 GridFS를 사용하여 파일을 저장
        const bucket = new GridFSBucket(db); // GridFSBucket 객체 생성

        const uploadStream = bucket.openUploadStream(filename); // 파일을 GridFS에 업로드할 스트림을 엶

        // 기존에 동일한 파일이 있는지 확인
        const existingFile = await db.collection(FILES_COLLECTION).findOne({ filehash: filehash });

        if (existingFile) {
            console.log('Existing file found with the same hash. Deleting old file...');

            // 기존 파일이 있으면 그 파일의 signature_id를 재사용
            signature_id = existingFile.signature_id;

            // 기존 파일 삭제 (메타데이터와 GridFS 파일 모두 삭제)
            await db.collection(FILES_COLLECTION).deleteOne({ _id: existingFile._id });
            await bucket.delete(existingFile.gridfs_file_id); // GridFS 파일 삭제
```

```
            await bucket.delete(existingFile.gridfs_file_id); // GridFS 파일 삭제

            console.log('Old file deleted successfully.');
        } else {
            // 기존 파일이 없으면 새로운 signature_id 생성
            signature_id = await generateSignatureId();
        }

        // 파일 데이터를 업로드하는 비동기 작업을 기다림
        await new Promise((resolve, reject) => {
            uploadStream.end(fileBuffer); // 업로드 스트림에 파일 데이터를 쓰고 끝냄
            uploadStream.on('finish', resolve); // 업로드가 끝나면 resolve 호출
            uploadStream.on('error', reject); // 업로드 중 에러가 발생하면 reject 호출
        });

        console.log('File uploaded successfully to GridFS');

        // 현재 UTC시간을 가져옴
        const now = new Date();
        now.setHours(now.getHours() + 9); // 9시간을 더해 KST(한국 표준시)로 변환

        // 업로드가 완료되면 파일 메타데이터를 MongoDB 컬렉션에 저장
        const fileMetadata = {
            signature_id: signature_id, // 고유한 signature_id
            filehash: filehash, // 파일의 MD5 해시값
            filename: filename, // 업로드된 파일의 이름
            gridfs_file_id: uploadStream.id, // GridFS에 저장된 파일의 ID
            upload_time: now, // 업로드 시간
            upload_ip: uploadIp, // 파일을 업로드한 사용자의 IP 주소
        };

        await db.collection(FILES_COLLECTION).insertOne(fileMetadata); // 파일 메타데이터를 'file' 컬렉

        // 2초 지연을 추가
        await delay(2000);

        // info 컬렉션에서 업로드된 파일의 MD5 해시값과 일치하는 문서를 찾음
        const infoCollection = db.collection('info'); // 'info' 컬렉션 선택
        const infoDocuments = await infoCollection.find({ md5: filehash }).toArray(); // MD5 값이 일치


        // 성공적인 응답을 클라이언트에 반환 (일치하는 문서와 함께)
        res.json({
            filename: `${filename}`, // 성공 메시지
            infoDocuments: infoDocuments, // 일치하는 문서들 반환
        });
    } catch (error) {
        // 파일 처리 중에 발생한 에러를 처리
        console.error('Error processing file upload:', error.message);
        console.error(error.stack); // 에러 스택을 출력하여 상세 정보를 확인
        res.status(500).json({ error: 'Error processing file upload', details: error.message });
    }
};

connectToMongoDB(); // 서버가 시작되면 MongoDB에 연결

// 다른 모듈에서 사용할 수 있도록 Multer와 업로드 처리 함수를 내보냄
export { scanUpload, scanUploadFile };
```

| | |
|---|---|
| 목적 | 사용자가 업로드한 파일을 MongoDB GridFS에 저장하고, MD5 해시값을 계산해 중복 파일을 처리. 기존 파일이 있으면 삭제 후 대체하며, 고유한 signature_id를 생성해 메타데이터를 저장. |

| | |
|---|---|
| | 이후 업로드된 파일의 해시와 일치하는 정보를 `info` 컬렉션에서 찾아 클라이언트에 반환. |
| 파일명 | uploadFileScan.js |

```
import { MongoClient, GridFSBucket } from 'mongodb';
import multer from 'multer'; // 파일 업로드 처리 모듈
import dotenv from 'dotenv'; // 환경 변수 로드를 위한 dotenv 모듈

dotenv.config(); // 환경 변수 로드

// MongoDB 연결 정보
const MONGO_URI = process.env.MONGO_URI; // .env 파일에서 MongoDB URI 로드
const FILES_COLLECTION = 'filedata'; // 파일 메타데이터가 저장될 컬렉션 이름
const PE_INFO_COLLECTION = 'pe_info'; // pe_info 컬렉션

let db, encryptedDb; // MongoDB 연결을 위한 변수

// Multer 설정: 메모리에서 파일을 처리하는 미들웨어
const packUpload = multer(); // Multer는 파일을 메모리 내에서 처리함

// MongoDB 연결 함수: 서버 시작 시 데이터베이스 연결 설정
const connectToMongoDB = async () => {
    try {
        const client = await MongoClient.connect(MONGO_URI);
        db = client.db('normal_files'); // normal_files DB
        encryptedDb = client.db('encrypted_files'); // encrypted_files DB
        console.log('uploadFilePack.js Connected to MongoDB');
    } catch (error) {
        console.error('MongoDB Connection Error:', error);
    }
};

// signature_id 생성 함수: 고유한 signature_id를 생성 (오늘 날짜 기반)
const generateSignatureId = async () => {
    const today = new Date().toISOString().slice(0, 10).replace(/-/g, '');
    const collection = db.collection(FILES_COLLECTION); // 'filedata' 컬렉션 선택

    // 오늘 날짜로 시작하는 signature_id의 개수를 확인하여 번호 매김
    const count = await collection.countDocuments({
        signature_id: { $regex: `^${today}-` }
    });
    const nextId = count + 1;

    // signature_id를 YYYYMMDD-001 형식으로 생성
    return `${today}-${nextId.toString().padStart(3, '0')}`;
};
```

```
43
44  // encrypted_Files의 filedata, pe_info 에서 signature_id를 기준으로 데이터를 조회하는 함수
45  const fetchMatchingFiledata = async (signature_id) => {
46      const encryptedFileInfo = await encryptedDb.collection(PE_INFO_COLLECTION).find({ signature_id }).toArray();
47      // encrypted_files에서 filedata 조회
48      const encryptedFilesData = await encryptedDb.collection(FILES_COLLECTION).find({ signature_id }).toArray();
49
50      return {
51          encryptedFilesData,
52          encryptedFileInfo,
53      };
54  };
55
56  // 암호화된 파일이 저장되었는지 확인하는 함수
57  const checkEncryptedFileUpload = async (signature_id, res, filename) => {
58      try {
59          const maxRetries = 600; // 최대 600번(600초) 재시도
60          let retries = 0;
61
62          const checkIfFileExists = async () => {
63              const encryptedFilesInfo = await encryptedDb.collection(PE_INFO_COLLECTION).find({ signature_id }).toArray();
64
65              if (encryptedFilesInfo.length > 0) {
66                  // pe_info에서 해당 signature_id에 해당하는 데이터 가져오기
67                  const peInfo = await fetchMatchingFiledata(signature_id);
68
69                  // 클라이언트에 성공 응답 반환
70                  res.json({
71                      message: 'File uploaded successfully',
72                      filename: filename,
73                      peInfo, // 조회된 pe_info 데이터
74                  });
75              } else if (retries < maxRetries) {
76                  // 파일이 아직 저장되지 않았으면 재시도
77                  retries++;
78                  console.log(`Retry ${retries}: Checking if file is uploaded...`);
79                  setTimeout(checkIfFileExists, 1000); // 1초 후 다시 체크
80              } else {
81                  // 최대 재시도 횟수 초과
82                  res.status(500).json({ message: 'File upload failed or timeout' });
83              }
84          };
85
86          checkIfFileExists(); // 파일 확인 시작
87
```

```
87
88      } catch (error) {
89          console.error('Error checking encrypted file upload:', error);
90          res.status(500).json({ error: 'Error checking encrypted file upload', details: error.message });
91      }
92  }
93
94  // 파일 업로드를 처리하는 함수: 사용자가 업로드한 파일을 MongoDB GridFS에 저장하는 함수
95  const packUploadFile = async (req, res) => {
96      try {
97          const fileBuffer = req.file.buffer; // 업로드된 파일의 데이터를 가져옴
98          const filename = req.file.originalname; // 파일의 이름을 가져옴
99          const uploadIp = req.ip; // 사용자의 IP 주소를 가져옴
100
101         // signature_id 생성
102         const signature_id = await generateSignatureId();
103
104         // GridFS에 파일 저장
105         const bucket = new GridFSBucket(db);
106         const uploadStream = bucket.openUploadStream(filename);
107
108         // 파일 업로드를 처리하는 비동기 작업
109         await new Promise((resolve, reject) => {
110             uploadStream.end(fileBuffer); // 파일 데이터를 스트림에 기록하고 업로드 종료
111             uploadStream.on('finish', () => {
112                 console.log('File uploaded to normal_files successfully.');
113                 resolve(); // 업로드 완료
114             });
115             uploadStream.on('error', reject); // 오류 발생 시 reject 호출
116         });
117
118         console.log('File uploaded successfully to GridFS');
119
120         // 현재 시간(UTC)을 KST로 변환
121         const now = new Date();
122         now.setHours(now.getHours() + 9); // 한국 시간 기준
123
124         // 파일의 메타데이터 저장
125         const fileMetadata = {
126             signature_id: signature_id, // 고유한 signature_id
127             filename: filename, // 업로드된 파일의 이름
128             file_extension: `.${filename.split('.').pop()}`, // 파일 확장자
129             gridfs_file_id: uploadStream.id, // GridFS에서 저장된 파일 ID
130             upload_time: now, // 업로드 시간
131             upload_ip: uploadIp // 파일을 업로드한 사용자의 IP 주소
132         };
133
134         // 메타데이터를 'filedata' 컬렉션에 삽입
135         await db.collection(FILES_COLLECTION).insertOne(fileMetadata);
136
137         // 암호화된 파일이 저장되었는지 확인
138         await checkEncryptedFileUpload(signature_id, res, filename); // encrypted_files에 파일 저장 확인
139
140     } catch (error) {
141         console.error('Error processing file upload:', error);
142         res.status(500).json({ error: 'Error processing file upload', details: error.message });
143     }
144 };
145
146 // MongoDB 연결
147 connectToMongoDB();
148
149 // 다른 모듈에서 사용할 수 있도록 Multer와 업로드 처리 함수를 내보냄
150 export { packUpload, packUploadFile };
```

| | |
|---|---|
| 목적 | 사용자가 업로드한 파일을 MongoDB의 GridFS에 저장하고, 고유한 signature_id를 생성해 메타데이터를 기록.<br>업로드 후 암호화된 파일의 저장 여부를 일정 시간 |

| | |
|---|---|
| | 동안 지속적으로 확인하여 클라이언트에 파일 정보를 반환.<br>파일의 메타데이터와 암호화 상태를 `filedata`와 `encrypted_files` 데이터베이스에 저장. |
| 파일명 | UploadFilePack.js |

```javascript
import { MongoClient, GridFSBucket, ObjectId } from 'mongodb';
import dotenv from 'dotenv'; // 환경 변수 설정
dotenv.config(); // 환경 변수 로드

// MongoDB 연결 정보
const MONGO_URI = process.env.MONGO_URI;
let encryptedDb; // MongoDB 연결을 위한 변수

// MongoDB 연결 함수
const connectToMongoDB = async () => {
    try {
        const client = await MongoClient.connect(MONGO_URI);
        encryptedDb = client.db('encrypted_files'); // 'encrypted_files' DB 사용
    } catch (error) {
        console.error('MongoDB Connection Error:', error);
    }
};

// GridFS에서 파일을 다운로드하는 함수
const downloadEncryptedFile = async (req, res) => {
    const fileId = req.params.fileId; // 클라이언트에서 넘어온 파일 ID

    try {
        const bucket = new GridFSBucket(encryptedDb); // GridFSBucket 객체 생성

        // 여기서 ObjectId 생성 시 new 키워드를 추가
        const downloadStream = bucket.openDownloadStream(new ObjectId(`${fileId}`));

        // 응답 헤더 설정: 다운로드할 파일의 MIME 타입과 이름 설정
        res.set({
            'Content-Type': 'application/octet-stream', // 기본 바이너리 파일로 설정
            'Content-Disposition': `attachment; filename="${fileId}"`, // 파일명 설정
        });

        // 파일 데이터를 클라이언트로 바로 전송
        downloadStream.pipe(res);

        // 파일을 찾을 수 없을 때
        downloadStream.on('error', (err) => {
            console.error('File not found:', err);
            res.status(500).send('File not found');
        });

        // 다운로드 완료 후
        downloadStream.on('end', () => {
            console.log(`File ${fileId} successfully downloaded.`);
        });

    } catch (error) {
        console.error('Failed to download file:', error);
        res.status(500).json({ error: 'Failed to download file' });
    }
};

// MongoDB 연결 호출
connectToMongoDB();

// export를 통해 다른 모듈에서 이 함수 사용 가능하게 함
export { downloadEncryptedFile };
```

| 목적 | 클라이언트가 요청한 파일 ID를 기반으로, 해당 파일을 GridFS에서 찾아 클라이언트로 스트리밍 |
|---|---|

| | |
|---|---|
| | 하여 다운로드함.<br>또한, 파일이 존재하지 않을 경우 에러를 처리하고, 파일 다운로드가 완료되면 로그를 기록. |
| 파일명 | DownloadFile.js |