4장

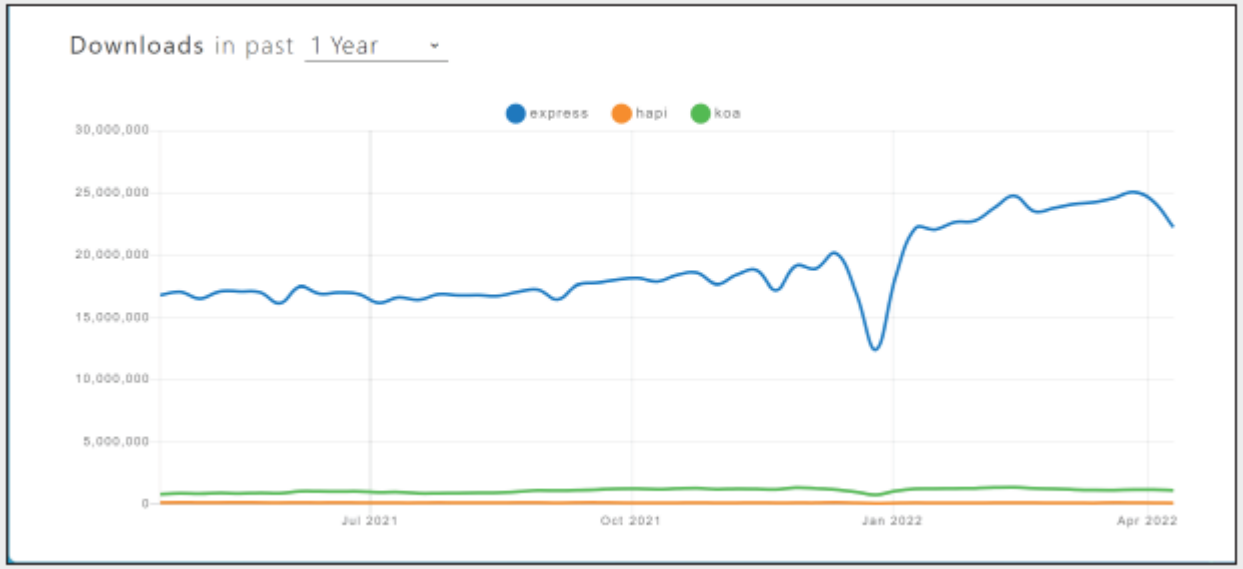
4장

4.1 익스프레스 프로젝트 시작하기

1. Express 소개

- http 모듈로 웹 서버를 만들 때 코드가 보기 좋지 않고, 확장성도 떨어짐
  - 프레임워크로 해결
  - 대표적인 것이 Express(익스프레스), Koa(코아), Hapi(하피)
  - 코드 관리도 용이하고 편의성이 많이 높아짐

그림 6-1 express, koa, hapi의 다운로드 수 비교



2. package.json 만들기

- 직접 만들거나 npm init 명령어 생성
  - node app으로 실행 가능

```
npm init

npm i express
npm i -D nodemon
```

  - 만약 nodemon app으로 실행하면,
    nodemon이 소스 코드 변경 시 서버를 재시작해줌
  - "start": "nodemon app" 추가하면,
    npm start 또는 npm run start으로 실행 가능

```json
{
  "name": "express_server",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "start": "nodemon app",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "choi",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.3"
  },
  "devDependencies": {
    "nodemon": "^3.1.0"
  }
}
```

3. app.js 작성하기

- 서버 구동의 핵심이 되는 파일
  - app.get('주소', 라우터)로 GET 요청이 올 때 어떤 동작을 할지 지정
  - app.listen('포트', 콜백)으로 몇 번 포트에서 서버를 실행할지 지정

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('hello express');
});

app.listen(8080, ()=>{
    console.log('익스프레스 서버 실행');
});
```

3. app.js 작성하기

- 서버 구동의 핵심이 되는 파일
  - app.set('port', 포트)로 서버가 실행될 포트 지정

```
const express = require('express');
const app = express();

app.set('port', process.env.PORT || 8080);

app.get('/', (req, res) => {
    res.send('hello express');
});

app.listen(app.get('port'), ()=>{
    console.log(app.get('port'), '익스프레스 서버 실행');
});
```

4. 익스프레스 서버 실행하기

- npm start (package.json의 start 스크립트) 콘솔에서 실행
- localhost:8080 또는 127.0.0.1:8080으로 접속

5. HTML 서빙하기

- res.sendFile을 이용하면 HTML 서빙 가능

```
<!DOCTYPE html>
<html lang="ko">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="...">
    <title>익스프레스 서버</title>
</head>
<body>
    <h1>Hello Express</h1>
    <p>나의 첫 익스프레스 서버</p>
</body>
</html>
```

```
const express = require('express');
const app = express();
const path = require('path')

app.set('port', process.env.PORT || 8080);

app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'index.html'));
});

app.listen(app.get('port'), ()=>{
    console.log(app.get('port'), '익스프레스 서버 실행');
});
```

6. 서버 실행하기

- app.js: 핵심 서버 스크립트
- public: 외부에서 접근 가능한 파일들 모아둠
- views: 템플릿 파일을 모아둠
- routes: 서버의 라우터와 로직을 모아둠
  - 추후에 models를 만들어 데이터베이스 사용

4.2 자주 사용하는 미들웨어

1. 미들웨어

```
const express = require('express');
const app = express();
const path = require('path')
app.set('port', process.env.PORT || 8080);

app.get('/', (req, res) => {
    console.log('모든 요청에 공통적으로 실행');
    res.sendFile(path.join(__dirname, 'index.html'));
});

app.get('/about', (req, res) => {
    console.log('모든 요청에 공통적으로 실행');
    res.send('hello express');
});

app.get('/service', (req, res) => {
    console.log('모든 요청에 공통적으로 실행');
    res.send('hello express');
});

app.listen(app.get('port'), ()=>{
    console.log(app.get('port'), '익스프레스 서버 실행');
});
```

1. 미들웨어

- 익스프레스는 미들웨어로 구성됨
  - 요청과 응답의 중간에 위치해 미들웨어 함수
  - app.use(path, 미들웨어[, 미들웨어...])
  - next()로 다음 미들웨어로 넘어감
  - 즉, 위에서 아래로 순서대로 실행된다. (순서가 중요)
  - 미들웨어는 req, res, next가 매개변수인 함수
  - req: 요청, res: 응답 조작 가능

표 6-1 미들웨어가 실행되는 경우

app.use(미들웨어)	모든 요청에서 미들웨어 실행
app.use('/abc', 미들웨어)	abc로 시작하는 요청에서 미들웨어 실행
app.post('/abc', 미들웨어)	abc로 시작하는 POST 요청에서 미들웨어 실행

```
const express = require('express');
const app = express();
const path = require('path')
app.set('port', process.env.PORT || 8080);

app.use((req, res, next) => {
    console.log('모든 요청에 공통적으로 실행');
    next();
});

app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'index.html'));
});

app.get('/about', (req, res) => {
    res.send('hello express');
});

app.get('/service', (req, res) => {
    res.send('hello express');
});

app.listen(app.get('port'), ()=>{
    console.log(app.get('port'), '익스프레스 서버 실행');
});
```

2. 에러 처리 미들웨어

- 에러가 발생하면 에러 처리 미들웨어!
  - 에러 처리 미들웨어 없이도 에러를 알아서 처리해준다.
  - err, req, res, next까지 매개변수가 4개
  - 첫 번째 err에는 에러가 관한 정보가 담김
  - res.status 메서드로 HTTP 상태 코드 지정 가능
  - 특별한 경우가 아니면 가장 아래에 위치

```
const path = require('path')
app.set('port', process.env.PORT || 8080);

app.use((req, res, next) => {
    console.log('모든 요청에 공통적으로 실행');
    next();
},(req, res, next) => {
    console.log(nonExistVariable);
    next();
});

app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'index.html'));
});

app.use((err, req, res, next) => {
    console.error(err);
    res.status(500).send('에러가 발생했습니다. 관리자에게 문의하세요.')
})

app.listen(app.get('port'), ()=>{
    console.log(app.get('port'), '익스프레스 서버 실행');
});
```

2. 에러 처리 미들웨어

- 404 에러는 에러 처리 미들웨어에서 에러로 인식하지 않는다.
  - 상태코드 404를 처리하기 위한 미들웨어를 추가
  - .status(에러코드)를 생략하면, 상태코드 200을 전달

```
const path = require('path')
app.set('port', process.env.PORT || 8080);
app.use((req, res, next) => {
    console.log('모든 요청에 공통적으로 실행');
    next();
},(req, res, next) => {
    console.log(nonExistVariable);
    next();
});
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'index.html'));
});
app.use((req, res, next) => {
    res.status(404).send('접근할 수 없는 경로');
});
app.use((err, req, res, next) => {
    console.error(err);
    res.status(500).send('에러가 발생했습니다. 관리자에게 문의하세요.')
})
app.listen(app.get('port'), ()=>{
    console.log(app.get('port'), '익스프레스 서버 실행');
});
```

3. 자주 쓰는 미들웨어

- morgan, cookie-parser, express-session dotenv 설치

```
npm i morgan cookie-parser express-session dotenv
```

  - app.use로 장착
  - 내부에서 알아서 next를 호출해서 다음 미들웨어로 넘어감

4. dotenv

- .env 파일을 읽어서 process.env로 만듦
  - 비밀키, 환경설정 변수 등을 따로 보관하는 용도
  - dot(점) + env
  - process.env.COOKIE_SECRET에 cookiesecret 값이 할당됨(키=값 형식)
  - 비밀 키들을 소스 코드에 그대로 적어두면 소스 코드가 유출되었을 때 비밀 키도 같이 유출됨
  - .env 파일에 비밀 키들을 모아두고 .env 파일만 잘 관리하면 됨

```
PORT=3001
SECRET_KEY=password
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=password
DB_DATABASE=my_db
```

```
require('dotenv').config();
```

5. morgan

- 서버로 들어온 요청과 응답을 기록해주는 미들웨어
  - 로그의 자세한 정도 선택 가능(dev, tiny, short, common, combined)
  - 예시) GET / 200 51.267 ms – 1539

  - 순서대로 HTTP요청 요청주소 상태코드 응답속도 – 응답바이트
  - 개발환경에서는 dev, 배포환경에서는 combined를 애용함.

```
const morgan = require('morgan');
app.use(morgan('dev'))
```

6. static

- 정적인 파일들을 제공하는 미들웨어
  - 인수로 정적 파일의 경로를 제공
  - 파일이 있을 때 fs.readFile로 직접 읽을 필요 없음
  - 요청하는 파일이 없으면 알아서 next를 호출해 다음 미들웨어로 넘어감
  - 파일을 발견했다면 다음 미들웨어는 실행되지 않음

```
app.use('요청경로', express.static('파일경로'));

const path = require('path')
app.use('/', express.static(path.join(__dirname, 'public')));
```

- 컨텐츠 요청 주소와 실제 컨텐츠의 경로를 다르게 만들 수 있음
  - 요청 주소 localhost:3000/stylesheets/style.css
  - 실제 컨텐츠 경로 /public/stylesheets/style.css
  - 서버의 구조를 파악하기 어려워져서 보안에 도움이 됨

9. json 미들웨어, urlencoded 미들웨어

- 요청의 본문을 해석해주는 미들웨어
  - 폼 데이터나 AJAX 요청의 데이터 처리
  - json 미들웨어는 요청 본문이 json인 경우 해석, urlencoded 미들웨어는 폼 요청 해석
  - put이나 patch, post 요청 시에 req.body에 프론트엔드에서 온 데이터가 들어감

```
app.use(express.json())

app.use(express.urlencoded({ extended: true }))
```

- Multipart 데이터(이미지, 동영상 등)인 경우는 다른 미들웨어(multer)를 사용

10. cookie-parser

- 요청 헤더의 쿠키를 해석해주는 미들웨어
  - parseCookies 함수와 기능이 비슷함
  - req.cookies 안에 쿠키들이 들어있음
  - 비밀키를 넣으면, 쿠키 뒤에 서명을 붙여 내 서버가 만든 쿠키임을 검증할 수 있음 (선택)

```
const cookieParser = require('cookie-parser');
app.use(cookieParser('비밀키'))
```

- 실제 쿠키 옵션들을 넣을 수 있음
  - expires, domain, httpOnly, maxAge, path, secure, sameSite 등
  - 지울 때는 clearCookie로(expires와 maxAge를 제외한 옵션들이 일치해야 함)

```
res.cookie('이름', encodeURIComponent('값'), {
    expires: new Date(만료기한),
    httpOnly: false,
    path: '/'
})
```

```
res.clearCookie('이름', encodeURIComponent('값'), {
        httpOnly: true,
        path: '/'
    })
```

11. express-session

- 세션 관리용 미들웨어
  - 세션 쿠키에 대한 설정 (secret: 쿠키 암호화, cookie: 세션 쿠키 옵션)
  - 세션 쿠키는 앞에 s%3A가 붙은 후 암호화되어 프런트에 전송됨
  - resave: 요청이 왔을 때 세션에 수정사항이 생기지 않아도 다시 저장할지 여부
  - saveUninitialized: 세션에 저장할 내역이 없더라도 세션을 저장할지
  - req.session.save로 수동 저장도 가능하지만 할 일 거의 없음

```
const session = require('express-session');

app.use(session({
    resave: false,
    saveUninitialized: false,
    secret: 'password',
    cookie: {
        httpOnly: true
    },
    name: 'connect.sid',
}))
```

```
req.session.name = '값';
req.sessionID;
req.session.destroy();
```

12. 미들웨어의 특성

- req, res, next를 매개변수로 가지는 함수

```
app.use((req, res, next) => {
    console.log('모든 요청에 공통적으로 실행');
    next();
});
```

- 익스프레스 미들웨어들도 다음과 같이 축약 가능
  - 순서가 매우 중요
  - static 미들웨어에서 파일을 찾으면 next를 호출하지 않음

```
app.use(
    morgan('dev'),
    express.static(path.join(__dirname, 'public')),
    express.json(),
    express.urlencoded({ extended: true }),
    cookieParser(),
    session({
        resave: false,
        saveUninitialized: false,
        secret: 'password',
        cookie: {
            httpOnly: true
        },
        name: 'connect.sid',
    })
)
```

13. next

- next를 호출해야 다음 코드로 넘어감
  - next를 주석 처리하면 응답이 전송되지 않음
  - 다음 미들웨어(라우터 미들웨어)로 넘어가지 않기 때문
  - next에 인수로 값을 넣으면 에러 핸들러로 넘어감('route'인 경우 다음 라우터로)

그림 6-6 next의 동작

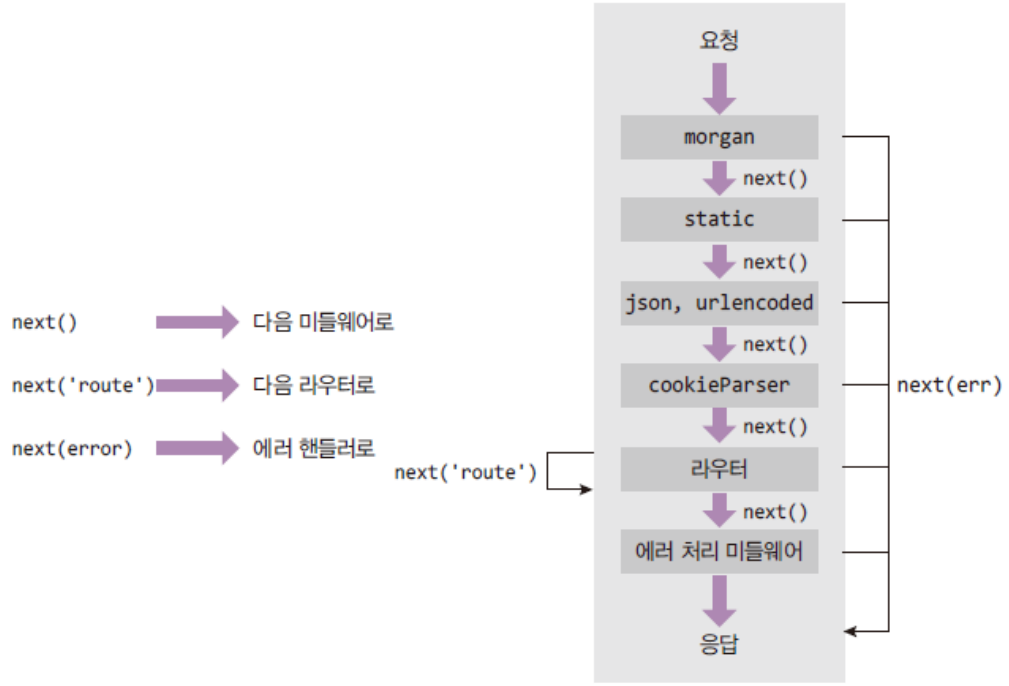


그림 6-7 에러 처리 미들웨어로 에러 보내기

```
next(err)

(err, req, res, next) => { }
```

14. 미들웨어 간 데이터 전달하기

- req나 res 객체 안에 값을 넣어 데이터 전달 가능
  - app.set과의 차이점: app.set은 서버 내내 유지, req, res는 요청하는 동안만 유지
  - 일반적으로 res.locals 객체를 데이터 전달용으로 많이 사용함

```
app.use((req, res, next) => {
    res.locals.data = '데이터 넣기';
    next();
}, (req, res, next) => {
    console.log(res.locals.data);
    next();
});
```

15. 미들웨어 확장하기

- 미들웨어 안에 미들웨어를 넣는 방법
  - 아래 두 코드는 동일한 역할

```
app.use(morgan('dev'))
```

```
app.use((req,res,next)=> {
    morgan('dev')(req, res, next);
})
```

  - 아래처럼 다양하게 활용 가능

```
app.use((req,res,next)=> {
    if (process.env.NODE_ENV === "development") {
        morgan('dev')(req, res, next)
    } else {
        morgan('combined')(req, res, next)
    }
})
```

16. 멀티파트 데이터 형식

- form 태그의 enctype이 multipart/form-data인 경우
  - 요청 본문을 해석할 수 없기에 multer 패키지가 필요

콘솔

```
$ npm i multer
```

multipart.html

```
<form action="/upload" method="post" enctype="multipart/form-data">
  <input type="file" name="image" />
  <input type="text" name="title" />
  <button type="submit">업로드</button>
</form>
```

17. multer 설정하기

multer 함수를 호출

- storage는 저장할 공간에 대한 정보
  - diskStorage는 하드디스크에 업로드 파일을 저장한다는 것
- destination은 저장할 경로를 done의 두 번째 인수로 넘기면 됨
- filename은 저장할 파일명(파일명+날짜+확장자 형식)을 done으로 넘기면 됨
- limits는 파일 개수나 파일 사이즈를 제한할 수 있음.

```
const multer = require('multer');

const upload = multer({
  storage: multer.diskStorage({
    destination(req, file, done) {
      done(null, 'uploads/');
    },
    filename(req, file, done) {
      const ext = path.extname(file.originalname);
      done(null, path.basename(file.originalname, ext) + Date.now() + ext);
    },
  }),
  limits: { fileSize: 5 * 1024 * 1024 },
});
```

- 실제 서버 운영 시에는 서버 디스크 대신에 S3같은 스토리지 서비스에 저장하는 게 좋음
  - storage 설정만 바꿔주면 됨

18. multer 미들웨어들

- single과 none, array, fields 미들웨어 존재
  - single은 하나의 파일을 업로드할 때, none은 파일을 아예 업로드하지 않을 때
  - req.file 안에 업로드 정보 저장

```
app.post('/upload', upload.single('image'), (req, res) => {
  console.log(req.file, req.body);
  res.send('ok');
});

app.post('/upload',  upload.none(),  (req, res) => {
  console.log(req.body);
  res.send('ok');
});
```

```
{
  fieldname: 'img',
  originalname: 'nodejs.png',
  encoding: '7bit',
  mimetype: 'image/png',
  destination: 'uploads/',
  filename: 'nodejs1514197844339.png',
  path: 'uploads\\nodejs1514197844339.png',
  size: 53357
}
```

  - array와 fields는 여러 개의 파일을 업로드 할 때 사용
  - array는 하나의 요청 body 이름 아래 여러 파일이 있는 경우
  - fields는 여러 개의 요청 body 이름 아래 파일이 하나씩 있는 경우
  - 두 경우 모두 업로드된 이미지 정보가 req.files 아래에 존재

```
app.post('/upload', upload.array('many'), (req, res) => {
  console.log(req.files, req.body);
  res.send('ok');
});
```

```
app.post('/upload',
  upload.fields([{ name: 'image1' }, { name: 'image2' }]),
  (req, res) => {
    console.log(req.files, req.body);
    res.send('ok');
  },
);
```

4.3 Router 객체로 라우터 분리하기

1. express.Router

- app.js가 길어지는 것을 막을 수 있음
  - userRouter의 get은 /user와 /가 합쳐져서 GET /user/가 됨

```
require('dotenv').config();
const express = require('express');
const app = express();

const indexRouter = require('./routes/index');
const usersRouter = require('./routes/users');

app.use('/', indexRouter);
app.use('/users', usersRouter);

app.use((req, res, next) => { // 404
    res.status(404).send('접근할 수 없는 경로');
})

app.use((err, req, res, next) => { // err
    console.error(err);
    res.status(500).send('에러가 발생했습니다.')
})
app.listen(app.get('port'), ()=>{
    console.log(app.get('port'), '서버 실행');
});
```

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

router.get('/', (req, res) => {
    res.send('Hello, Index');
});

module.exports = router;
```

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

router.get('/', (req, res) => {
    res.send('Hello, User');
});

module.exports = router;
```

2. 라우트 매개변수

» :id를 넣으면 req.params.id로 받을 수 있음

- 동적으로 변하는 부분을 라우트 매개변수로 만듦

```
router.get('/:id', (req, res) => {
    console.log(req.params, req.query);
    res.send(`Hello, User${req.params.id}`)
});
```

- 일반 라우터보다 뒤에 위치해야 함

```
router.get('/:id', (req, res) => {
    console.log(req.params, req.query);
    res.send(`Hello, User ${req.params.id}`)
});

router.get('/detail', (req, res) => {
    res.send("순서를 바꾸지 않으면, 실행되지 않는다.")
});
```

- /users/123?limit=5&skip=10 주소 요청인 경우

```
{ id: '123' } { limit: '5', skip: '10' }
GET /users/123?limit=5&skip=10 200 11.827 ms - 15
```

3. 404 미들웨어

- 요청과 일치하는 라우터가 없는 경우를 대비해 404 라우터를 만들기

```
app.use((req, res, next) => { // 404
    res.status(404).send('접근할 수 없는 경로');
});
```

  - 이게 없으면 단순히 Cannot GET 주소 라는 문자열이 표시됨

4. 라우터 그룹화하기

- 주소는 같지만 메서드가 다른 코드가 있을 때

```
router.get('/', (req, res) => {
    res.send('GET 요청: Hello, User');
});

router.post('/', (req, res) => {
    res.send('POST 요청: User 등록');
});
```

- router.route로 묶음

```
router.route('/')
    .get((req, res) => {
        res.send('GET 요청: Hello, User');
    })
    .post((req, res) => {
        res.send('POST 요청: User 등록');
    });
```

4.4 req, res 객체 살펴보기

1. req

- **req.app**: req 객체를 통해 app 객체에 접근 (req.app.get('port')와 같은 식으로 사용)
- **req.body**: body-parser 미들웨어가 만드는 요청의 본문을 해석한 객체
- **req.cookies**: cookie-parser 미들웨어가 만드는 요청의 쿠키를 해석한 객체
- **req.signedCookies**: cookie-parser 미들웨어가 만드는 요청 중 서명된 쿠키를 해석한 객체
- **req.ip**: 요청의 ip 주소
- **req.params**: 라우트 매개변수에 대한 정보가 담긴 객체
- **req.query**: 쿼리스트링에 대한 정보가 담긴 객체
- **req.get(헤더 이름)**: 헤더의 값을 가져오고 싶을 때 사용하는 메서드

2. res

- **res.app**: req.app처럼 res 객체를 통해 app 객체에 접근
- **res.cookie(키, 값, 옵션)**: 쿠키를 설정하는 메서드
- **res.clearCookie(키, 값, 옵션)**: 쿠키를 제거하는 메서드
- **res.setHeader(헤더, 값)**: 응답의 헤더를 설정
- **res.status(코드)**: 응답 시의 HTTP 상태 코드를 지정

- **res.end()**: 데이터 없이 응답 전송
- **res.json(JSON)**: JSON 형식의 응답 전송
- **res.redirect(주소)**: 리다이렉트할 주소와 함께 응답 전송
- **res.render(뷰, 데이터)**: 다음 절에서 다룰 템플릿 엔진을 렌더링해서 응답할 때 사용하는 메서드
- **res.send(데이터)**: 데이터와 함께 응답을 전송 (데이터는 문자열, HTML, 버퍼, 객체, 배열 가능)
- **res.sendFile(경로)**: 경로에 위치한 파일로 응답을 전송

3. 기타

- 메서드체이닝을 지원

```
router.get('/', (req, res) => {
    res
        .status(200)
        .cookie('my_cookie', 'cookie')
        .send('Hello')
});
```

- 응답은 한 번만!

```
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are sent to the client
    at ServerResponse.setHeader (node:_http_outgoing:652:11)
```

4.5 템플릿 엔진 사용하기

1. 템플릿 엔진

- HTML의 정적인 단점을 개선
  - 반복문, 조건문, 변수 등을 사용할 수 있음
  - 동적인 페이지 작성 가능
  - PHP, JSP와 유사

2. Pug(구 Jade)

- 문법이 Ruby와 비슷해 코드 양이 많이 줄어듦
  - HTML과 많이 달라 호불호가 갈림
  - 익스프레스에 app.set으로 퍼그 연결

```
express = require('express');
app = express();
path = require('path');

app.set('view engine', 'pug');
app.set('views', path.join(__dirname, 'views'));
```

그림 6-11 퍼그 로고

npm i pug

3. Pug – HTML 표현

퍼그	HTML

doctype html
html
  head
  title= title
  link(rel='stylesheet', href='/stylesheets/style.css')
	<!DOCTYPE html>
	<html>
	  <head>
	  <title>익스프레스</title>
	  <link rel="stylesheet" href="/style.css" />
	  </head>
	</html>

퍼그	HTML
#login-button .post-image span#highlight p.hidden.full	

퍼그	HTML
p Welcome to Express button(type='submit') 전송	Welcome to Express <button type="submit">전송</button>

4. Pug – HTML 표현

퍼그	HTML

p
  ¦ 안녕하세요.
  ¦ 여러 줄을 입력합니다.
  br
  ¦ 태그도 중간에 넣을 수 있습니다.
	
	  안녕하세요. 여러 줄을 입력합니다.
	
	  태그도 중간에 넣을 수 있습니다.

퍼그	HTML

style.
  h1 {
  font-size: 30px;
  }
script.
  const message = 'Pug';
  alert(message);
	<style>
	  h1 {
	  font-size: 30px;
	  }
	</style>
	<script>
	  const message = 'Pug';
	  alert(message);
	</script>

5. Pug - 변수

- res.render에서 두 번째 인수 객체에 Pug 변수를 넣음

```
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

  - res.locals 객체에 넣는 것도 가능(미들웨어간 공유됨)

```
router.get('/', function(req, res, next) {
  res.locals.title = 'Express';
  res.render('index');
});
```

  - =이나 #{}으로 변수 렌더링 가능(= 뒤에는 자바스크립트 문법 사용 가능)

퍼그	HTML
h1= title p Welcome to #{title} button(class=title, type='submit') 전송 input(placeholder=title + ' 연습')	Express Welcome to Express <button class="Express" type="submit">전송</button> <input placeholder="Express 연습" />

6. Pug – 파일 내 변수

- 퍼그 파일 안에서 변수 선언 가능
  - - 뒤에 자바스크립트 사용

퍼그	HTML
- const node = 'Node.js' - const js = 'Javascript' p # {node}와 # {js}	Node.js와 Javascript

  - 변수 값을 이스케이프 하지 않을 수도 있음(자동 이스케이프)

퍼그	HTML
p= '이스케이프' p!= '이스케이프하지 않음'	이스케이프 이스케이프하지 않음

7. Pug – 반복문

» for in이나 each in으로 반복문 사용

퍼그	HTML

ul
  each fruit in ['사과', '배', '오렌지', '바나나', '복숭아']
  li= fruit
	
	   - 사과
	   - 배
	   - 오렌지
	   - 바나나
	   - 복숭아

- 값과 인덱스 가져올 수 있음

퍼그	HTML

ul
  each fruit, index in ['사과', '배', '오렌지', '바나나', '복숭아']
  li= (index + 1) + '번째 ' + fruit
	
	    1번째 사과
	    2번째 배
	    3번째 오렌지
	    4번째 바나나
	    5번째 복숭아

8. Pug – 조건문

- if else if else문, case when문 사용 가능

퍼그	HTML

if isLoggedIn
  div 로그인 되었습니다.
else
  div 로그인이 필요합니다.
	<!-- isLoggedIn이 true일 때 -->
	 로그인 되었습니다.
	<!-- isLoggedIn이 false일 때 -->
	 로그인이 필요합니다.

퍼그	HTML

case fruit
  when 'apple'
  p 사과입니다.
  when 'banana'
  p 바나나입니다.
  when 'orange'
  p 오렌지입니다.
  default
  p 사과도 바나나도 오렌지도 아닙니다.
	<!-- fruit이 apple일 때 -->
	 사과입니다.
	<!-- fruit이 banana일 때 -->
	 바나나입니다.
	<!-- fruit이 orange일 때 -->
	 오렌지입니다.
	<!-- 기본값 -->
	 사과도 바나나도 오렌지도 아닙니다.

9. Pug – include

- 퍼그 파일에 다른 퍼그 파일을 넣을 수 있음
  - 헤더, 푸터, 내비게이션 등의 공통 부분을 따로 관리할 수 있어 편리
  - include로 파일 경로 지정

header.pug

```
header
  a(href='/') Home
  a(href='/about') About
```

footer.pug

```
footer
  div 푸터입니다
```

main.pug

```
include header
main
  h1 메인 파일
  p 다른 파일을 include할 수 있습니다.
include footer
```

HTML

```
<header>
  <a href="/">Home</a>
  <a href="/about">About</a>
</header>
<main>
  <h1>메인 파일</h1>
  <p>다른 파일을 include할 수 있습니다.</p>
</main>
<footer>
  <div>푸터입니다.</div>
</footer>
```

10. Pug – extends와 block

- 레이아웃을 정할 수 있음
  - 공통되는 레이아웃을 따로 관리할 수 있어 좋음, include와도 같이 사용

펑그	HTML

layout.pug

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/style.
css')
    block style
  body
    header 헤더입니다.
    block content
    footer 푸터입니다.
    block script
```

body.pug

```
extends layout

block content
  main
    p 내용입니다.

block script
  script(src="/main.js")
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Express</title>
    <link rel="stylesheet" href="/style.css"
/>
  </head>
  <body>
    <header>헤더입니다.</header>
    <main>
      <p>내용입니다.</p>
    </main>
    <footer>푸터입니다.</footer>
    <script src="/main.js"></script>
  </body>
</html>
```

11. 넌적스

- Pug의 문법에 적응되지 않는다면 넌적스를 사용
  - Pug를 지우고 Nunjucks 설치
  - 확장자는 html 또는 njk(view engine을 njk로)

npm i nunjucks

```
express = require('express');
app = express();
path = require('path');
nunjucks = require('nunjucks');
nunjucks.configure('views', {
    express: app,
    watch: true,
    autoescape: true
});

app.set('view engine', 'html');
app.set('views', path.join(__dirname, 'views'));
```

12. 넌적스 - 변수

- {{변수}}

넌적스

```
<h1>{{title}}</h1>
<p>Welcome to {{title}}</p>
<button class="{{title}}" type="submit">전송</button>
<input placeholder="{{title}} 연습" />
```

- 내부 변수 선언 가능 {%set 자바스크립트 구문 }

넌적스	HTML
{% set node = 'Node.js' %} {% set js = 'Javascript' %} {{node}}와 {{js}}	Node.js와 Javascript

넌적스	HTML
{{'이스케이프'}}	이스케이프
{{'이스케이프하지 않음' | safe }}	이스케이프하지 않음

13. 넌적스 - 반복문

- {% %} 안에 for in 작성(인덱스는 loop 키워드)

넌적스	HTML
 {% set fruits = ['사과', '배', '오렌지', '바나나', '복숭아'] %} {% for item in fruits %} {{item}} {% endfor %} 	 - 사과 - 배 - 오렌지 - 바나나 - 복숭아

넌적스	HTML
 {% set fruits = ['사과', '배', '오렌지', '바나나', '복숭아'] %} {% for item in fruits %} {{loop.index}}번째 {{item}} {% endfor %} 	 1번째 사과 2번째 배 3번째 오렌지 4번째 바나나 5번째 복숭아

14. 넌적스 - 조건문

» {% if %} 안에 조건문 작성

넌적스	HTML
{% if isLoggedIn %} 로그인 되었습니다. {% else %} 로그인이 필요합니다. {% endif %}	<!-- isLoggedIn이 true일 때 --> 로그인 되었습니다. <!-- isLoggedIn이 false일 때 --> 로그인이 필요합니다.

넌적스	HTML
{% if fruit === 'apple' %} 사과입니다. {% elif fruit === 'banana' %} 바나나입니다. {% elif fruit === 'orange' %} 오렌지입니다. {% else %} 사과도 바나나도 오렌지도 아닙니다. {% endif %}	<!-- fruit이 apple일 때 --> 사과입니다. <!-- fruit이 banana일 때 --> 바나나입니다. <!-- fruit이 orange일 때 --> 오렌지입니다. <!-- 기본값 --> 사과도 바나나도 오렌지도 아닙니다.

15. 넌적스 - include

- 파일이 다른 파일을 불러올 수 있음
  - include에 파일 경로 넣어줄 수 있음

넌적스	HTML

header.html
`<header>`
`  <a href="/">Home</a>`
`  <a href="/about">About</a>`
`</header>`
 	
footer.html
`<footer>`
`  <div>푸터입니다.</div>`
`</footer>`
 	
main.html
`{% include "header.html" %}`
`<main>`
`  <h1>메인 파일</h1>`
`  <p>다른 파일을 include할 수 있습니다.</p>`
`</main>`
`{% include "footer.html" %}`
	`<header>`
	`  <a href="/">Home</a>`
	`  <a href="/about">About</a>`
	`</header>`
	`<main>`
	`  <h1>메인 파일</h1>`
	`  <p>다른 파일을 include할 수 있습니다.</p>`
	`</main>`
	`<footer>`
	`  <div>푸터입니다.</div>`
	`</footer>`

16. 넌적스 - 레이아웃

- 레이아웃을 정할 수 있음
  - 공통되는 레이아웃을 따로 관리할 수 있어 좋음, include와도 같이 사용

넌적스

layout.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{title}}</title>
    <link rel="stylesheet" href="/style.
css" />
    {% block style %}
    {% endblock %}
  </head>
  <body>
    <header>헤더입니다.</header>
    {% block content %}
    {% endblock %}
    <footer>푸터입니다.</footer>
    {% block script %}
    {% endblock %}
  </body>
</html>
```

body.html

```
{% extends 'layout.html' %}

{% block content %}
<main>
  <p>내용입니다.</p>
</main>
{% endblock %}

{% block script %}
<script src="/main.js"></script>
{% endblock %}
```

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Express</title>
    <link rel="stylesheet" href="/style.css"
/>
  </head>
  <body>
    <header>헤더입니다.</header>
    <main>
      <p>내용입니다.</p>
    </main>
    <footer>푸터입니다.</footer>
    <script src="/main.js"></script>
  </body>
</html>
```

17. 에러 처리 미들웨어

- 에러 발생 시 템플릿 엔진과 상관없이 템플릿 엔진 변수를 설정하고 error 템플릿을 렌더링함
  - res.locals.변수명으로도 템플릿 엔진 변수 생성 가능
  - process.env.NODE_ENV는 개발환경인지 배포환경인지 구분해주는 속성

```
app.use((req, res, next) => {
    const error = new Error(`${req.method} ${req.url} 라우터가 없습니다.`);
    error.status(404);
    next(error);
});

app.use((err, req, res, next) => {
    res.locals.message = err.message;
    res.locals.error = process.env.NODE_ENV == 'development' ? err : {};
    res.status(err.status || 500);
    res.render('error');
});
```

4.6 데이터베이스

1. 데이터베이스란

- 지금까지는 데이터를 서버 메모리에 저장했음
  - 서버를 재시작하면 데이터도 사라져버림 -> 영구적으로 저장할 공간 필요
- MySQL 관계형 데이터베이스 사용
  - 데이터베이스: 관련성을 가지며 중복이 없는 데이터들의 집합
  - DBMS: 데이터베이스를 관리하는 시스템
  - RDBMS: 관계형 데이터베이스를 관리하는 시스템
  - 서버의 하드 디스크나 SSD 등의 저장 매체에 데이터를 저장
  - 서버 종료 여부와 상관 없이 데이터를 계속 사용할 수 있음
  - 여러 사람이 동시에 접근할 수 있고, 권한을 따로 줄 수 있음

그림 7-2 데이터베이스는 흔히 원기둥 세 개를 겹친 모양으로 표현합니다.

4.7 데이터베이스, 테이블 생성하기

1. 데이터베이스 생성하기

- 콘솔에서 MySQL 프롬프트에 접속
  - CREATE SCHEMA nodejs;로 nodejs 데이터베이스 생성
    - Character set은 utf8mbt(이모티콘 지원)
    - Collate는 Character set을 어떻게 정렬할 지에 관한 것
  - use nodejs;로 생성한 데이터베이스 선택

```
CREATE SCHEMA `nodejs` DEFAULT CHARACTER SET UTF8MB4 DEFAULT COLLATE UTF8MB4_GENERAL_CI;

USE `nodejs`;
```

2. 테이블 생성하기

- MySQL 프롬프트에서 테이블 생성
  - CREATE TABLE [데이터베이스명.테이블명]으로 테이블 생성
  - 사용자 정보를 저장하는 테이블

```
CREATE TABLE `Users` (
        `userId` VARCHAR(20) NOT NULL,
        `name` VARCHAR(20) NOT NULL,
        `gender` ENUM('M', 'F') NOT NULL,
        `password` VARCHAR(100) NOT NULL,
        `profileImage` VARCHAR(200) NOT NULL DEFAULT 'default.png',
        `createdAt` DATETIME NOT NULL,
        `updatedAt` DATETIME NOT NULL,
        PRIMARY KEY (`userId`)
    ) ENGINE = InnoDB;


CREATE TABLE `UserDetails` (
        `id` INTEGER NOT NULL auto_increment,
        `email` VARCHAR(100) NOT NULL,
        `phone` VARCHAR(20) NOT NULL,
        `country` VARCHAR(20) NOT NULL,
        `userId` VARCHAR(20),
        PRIMARY KEY (`id`),
        FOREIGN KEY (`userId`) REFERENCES `Users` (`userId`) ON DELETE
        SET NULL ON UPDATE CASCADE
    ) ENGINE = InnoDB;
```

3. 컬럼과 로우

- 나이, 결혼 여부, 성별같은 정보가 컬럼
- 실제로 들어가는 데이터는 로우

Note ≡ 컬럼과 로우

그림 7-27 컬럼과 로우

4. 컬럼 옵션들

- id INT NOT NULL AUTO_INCREMENT
  - 컬럼명 옆의 것들은 컬럼에 대한 옵션들

  - INT: 정수 자료형(FLOAT, DOUBLE은 실수)
  - VARCHAR: 문자열 자료형, 가변 길이(CHAR은 고정 길이)
  - TEXT: 긴 문자열은 TEXT로 별도 저장
  - DATETIME: 날짜 자료형 저장
  - TINYINT: -128에서 127까지 저장하지만 여기서는 1 또는 0만 저장해 불 값 표현

  - NOT NULL: 빈 값은 받지 않는다는 뜻(NULL은 빈 값 허용)
  - AUTO_INCREMENT: 숫자 자료형인 경우 다음 로우가 저장될 때 자동으로 1 증가
  - UNSIGNED: 0과 양수만 허용
  - ZEROFILL: 숫자의 자리 수가 고정된 경우 빈 자리에 0을 넣음
  - DEFAULT now(): 날짜 컬럼의 기본값을 현재 시간으로

5. Primary Key, Unique Index

- PRIMARY KEY(id)
  - id가 테이블에서 로우를 특정할 수 있게 해주는 고유한 값임을 의미
  - 학번, 주민등록번호같은 개념

- UNIQUE INDEX name_UNIQUE (name ASC)
  - 해당 컬럼(name)이 고유해야 함을 나타내는 옵션
  - name_UNIQUE는 이 옵션의 이름(아무거나 다른 걸로 지어도 됨)
  - ASC는 인덱스를 오름차순으로 저장함의 의미(내림차순은 DESC)

6. 테이블 옵션

- COMMENT: 테이블에 대한 보충 설명(필수 아님)
- ENGINE: InnoDB 사용(이외에 MyISAM이 있음, 엔진별로 기능 차이 존재)

7. 테이블 생성되었나 확인하기

- DESC 테이블명

```
DESC `Users`;
DESC `UserDetails`;
```

- 테이블 삭제하기: DROP TABLE 테이블명

```
DROP TABLE `users`;
DROP TABLE `UserDetails`;
```

8. 게시물 테이블 저장하기

Posts, HashTags, PostHashTags 테이블 생성

```
CREATE TABLE `Posts` (
        `id` INTEGER NOT NULL auto_increment,
        `content` TEXT NOT NULL,
        `createdAt` DATETIME NOT NULL,
        `updatedAt` DATETIME NOT NULL,
        `userId` VARCHAR(20),
        PRIMARY KEY (`id`),
        FOREIGN KEY (`userId`) REFERENCES `Users` (`userId`) ON DELETE
        SET NULL ON UPDATE CASCADE
    ) ENGINE = InnoDB;

CREATE TABLE `HashTags` (
        `id` INTEGER NOT NULL auto_increment,
        `name` VARCHAR(15) NOT NULL UNIQUE,
        PRIMARY KEY (`id`)
    ) ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS `PostHashTags` (
        `PostId` INTEGER,
        `HashTagId` INTEGER,
        PRIMARY KEY (`PostId`, `HashTagId`),
        FOREIGN KEY (`PostId`) REFERENCES `Posts` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
        FOREIGN KEY (`HashTagId`) REFERENCES `HashTags` (`id`) ON DELETE CASCADE ON UPDATE CASCADE
    ) ENGINE = InnoDB;
```

9. 외래키(foreign key)

- 댓글 테이블은 사용자 테이블과 관계가 있음(사용자가 댓글을 달기 때문)
  - 외래키를 두어 두 테이블이 관계가 있다는 것을 표시

  - FOREIGN KEY (컬럼명) REFERENCES 데이터베이스.테이블명 (컬럼)
  - FOREIGN KEY (commenter) REFERENCES nodejs.users (id)
  - 댓글 테이블에는 commenter 컬럼이 생기고 사용자 테이블의 id값이 저장됨

  - ON DELETE CASCADE, ON UPDATE CASCADE
  - 사용자 테이블의 로우가 지워지고 수정될 때 댓글 테이블의 연관된 로우들도 같이 지워지고 수정됨
  - 데이터를 일치시키기 위해 사용하는 옵션(CASCADE 대신 SET NULL과 NO ACTION도 있음)

10. 테이블 목록 보기

- SHOW TABLES;

```
SHOW TABLES;
```

4.8 CRUD 작업하기

1. CRUD

- Create, Read, Update, Delete로, 데이터베이스에서 많이 하는 작업 4가지

그림 7-37 CRUD 작업

2. Create

- INSERT INTO 테이블 (컬럼명들) VALUES (값들)

```
INSERT INTO `Users` (userId, name, gender, password, profileImage, createdAt, updatedAt)
VALUES('dlwlrma', '아이유', 'F', '1234', 'iu.png', now(), now());

INSERT INTO `UserDetails` (email, phone, country, userId)
VALUES('iu@gmail.com', '010-1234-1234', 'KOR', 'dlwlrma');


INSERT INTO posts (content, createdAt, updatedAt, userId)
VALUES ('안녕하세요 아이유입니다.', now(), now(), 'dlwlrma');
SET @postId = LAST_INSERT_ID();

INSERT INTO hashtags (name) VALUES ('인사말');
SET @hashtagId1 = LAST_INSERT_ID();

INSERT INTO hashtags (name) VALUES ('아이유');
SET @hashtagId2 = LAST_INSERT_ID();

INSERT INTO posthashtags (PostId, HashTagId)
VALUES (@postId, @hashtagId1), (@postId, @hashtagId2);
```

3. Read

- SELECT 컬럼 FROM 테이블명
  - SELECT * 은 모든 컬럼을 선택한다는 의미

```
SELECT * FROM `Users`;
```

  - 컬럼만 따로 추리는 것도 가능

```
SELECT id, userid, password FROM `Users`;
```

4. Read 옵션들

- WHERE로 조건을 주어 선택 가능
  - AND로 여러가지 조건을 동시에 만족하는 것을 찾음

```
SELECT id, userid, password
FROM `Users`
WHERE userid = 'dlwlrma' AND gender = 'F';
```

  - OR로 여러가지 조건 중 하나 이상을 만족하는 것을 찾음

```
SELECT id, userid, password
FROM `Users`
WHERE userid = 'dlwlrma' OR gender = 'F';
```

5. 정렬해서 찾기

- ORDER BY로 특정 컬럼 값 순서대로 정렬 가능
  - DESC는 내림차순, ASC 오름차순

```
SELECT * FROM `Posts` ORDER BY `id` ASC;

SELECT * FROM `Posts` ORDER BY `id` DESC;
```

6. LIMIT, OFFSET

- LIMIT으로 조회할 개수 제한

```
SELECT * FROM `Posts` ORDER BY `id` DESC LIMIT 1;
```

- OFFSET으로 앞의 로우들 스킵 가능(OFFSET 2면 세 번째 것부터 찾음)

```
SELECT * FROM `Posts` ORDER BY `id` DESC LIMIT 1 OFFSET 1;
```

7. Update

- 데이터베이스에 있는 데이터를 수정하는 작업
  - UPDATE 테이블명 SET 컬럼=새값 WHERE 조건

```
UPDATE `Users` SET `name` = '이지은' WHERE `userid` = 'dlwlrma';
```

8. Delete

- 데이터베이스에 있는 데이터를 삭제하는 작업
  - DELETE FROM 테이블명 WHERE 조건

```
DELETE FROM `Posts` WHERE `id` = 2;
```

4.9 시퀄라이즈 사용하기

1. 시퀄라이즈 ORM

- SQL 작업을 쉽게 할 수 있도록 도와주는 라이브러리
  - ORM: Object Relational Mapping: 객체와 데이터를 매핑(1대1 짝지음)
  - MySQL 외에도 다른 RDB(Maria, Postgre, SQLite, MSSQL)와도 호환됨
  - 자바스크립트 문법으로 데이터베이스 조작 가능
  - 프로젝트 세팅 후, 콘솔을 통해 경로로 이동한 후 package.json 작성

```
{
    "name": "sequelize",
    "version": "1.0.0",
    "description": "시퀄라이즈 학습",
    "main": "app.js",
    "scripts": {
    "start": "nodemon app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
},
    "author": "최인규",
    "license": "ISC"
}
```

2. 시퀄라이즈 CLI 사용하기

- 시퀄라이즈 명령어 사용하기 위해 sequelize-cli 설치
  - mysql2는 MySQL DB가 아닌 드라이버(Node.js와 MySQL을 이어주는 역할)

```
npm i express morgan nunjucks dotenv sequelize sequelize-cli mysql2
npm i -D nodemon
```

- npx sequelize init으로 시퀄라이즈 구조 생성

```
npx sequelize init
```

3. models/index.js 수정

- 다음과 같이 수정
  - require(../config/config) 설정 로딩
  - new Sequelize(옵션들…)로 DB와 연결 가능

```
'use strict';
const Sequelize = require('sequelize');
const process = require('process');
const env = process.env.NODE_ENV || 'development';
const config = require(__dirname + '/../config/config.json')[env];
const db = {};

let sequelize;
if (config.use_env_variable) {
  sequelize = new Sequelize(process.env[config.use_env_variable], config);
} else {
  sequelize = new Sequelize(config.database, config.username, config.password, config);
}

db.sequelize = sequelize;
db.Sequelize = Sequelize;

module.exports = db;
```

4. MySQL 연결하기

app.js 작성

- sequelize.sync로 연결

```
require('dotenv').config();
const express = require('express');
const path = require('path');
const morgan = require('morgan');
const nunjucks = require('nunjucks');
const { sequelize } = require('./models');
const app = express();
app.set('port', process.env.PORT || 8000);
app.set('view engine', 'html');
nunjucks.configure('views', {
    express: app,
    watch: true,
});

sequelize.sync({ force: false })
    .then(() => {
        console.log('데이터베이스 연결 성공');
    })
    .catch((err) => {
        console.error(err);
    });
```

```
app.use(
    morgan('dev'),
    express.static(path.join(__dirname, 'public')),
    express.json(),
    express.urlencoded({ extended: false })
);
app.use((req, res, next) => {
    const error = new Error(`${req.method} ${req.url} 라우터가 없습니다.`);
    error.status = 404;
    next(error);
});
app.use((err, req, res, next) => {
    res.locals.message = err.message;
    res.locals.error = process.env.NODE_ENV == 'development' ? err : {};
    res.status(err.status || 500);
    res.render('error');
});
app.listen(app.get('port'), () => {
    console.log(app.get('port'), '번 포트에서 대기 중');
});
```

5. config.json 설정하기

- DB 연결 정보를 넣기

```
{
    "development": {
        "username": "root",
        "password": 비밀번호,
        "database": "nodejs",
        "host": "127.0.0.1",
        "dialect": "mysql"
    },
    ...
}
```

6. 연결 테스트하기

- npm start로 실행해서 SELECT 1+1 AS RESULT가 나오면 연결 성공

```
$ npm  start

> sequelize@1.0.0 start
> nodemon app.js

[nodemon] 3.1.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
3001 번 포트에서 대기 중
Executing (default): SELECT 1+1 AS result
데이터베이스 연결 성공
```

7. 모델 생성하기

- 테이블에 대응되는 시퀄라이즈 모델 생성
  - static initiate가 모델을 시퀄라이즈와 연결
  - static associate는 모델 간 관계 설정

```
const Sequelize = require('sequelize');
class User extends Sequelize.Model {
    static initiate(sequelize) {
        User.init({
            userId: {...},
            name: {...},
            gender: {...},
            password: {...},
            profileImage: {...},
        }, {
            sequelize,
            timestamps: true,
            underscored: false,
            modelName: 'User',
            tableName: 'Users',
            paranoid: false,
            charset: 'utf8mb4',
            collate: 'utf8mb4_general_ci',
        });
    }
    static associate(db) {
        db.User.hasMany(db.Post, { foreignKey: 'userId', sourceKey: 'userId' });
        db.User.hasOne(db.UserDetail, { foreignKey: 'userId', sourceKey: 'userId' });
    }
}
module.exports = User;
```

8. 모델 옵션들

- 시퀄라이즈 모델의 자료형은 MySQL의 자료형과 조금 다름

표 7-1 MySQL과 시퀄라이즈의 비교

MySQL	시퀄라이즈
VARCHAR(100)	STRING(100)
INT	INTEGER
TINYINT	BOOLEAN
DATETIME	DATE
INT UNSIGNED	INTEGER.UNSIGNED
NOT NULL	allowNull: false
UNIQUE	unique: true
DEFAULT now()	defaultValue: Sequelize.NOW

- define 메서드의 세 번째 인자는 테이블 옵션
  - timestamps : true면 createdAt(생성 시간), updatedAt(수정 시간) 컬럼을 자동으로 생성
  - paranoid : true면 deletedAt(삭제 시간) 컬럼을 자동으로 생성 (soft delete)
  - underscored : 카멜케이스로 생성되는 컬럼을 스네이크케이스로 생성
  - modelName : 모델 이름
  - tableName : 테이블 이름을 설정
  - charset과 collate는 한글 설정을 위해 필요(이모티콘 넣으려면 utf8mb4로)

9. 게시글 모델 생성하기

- post.js 생성

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

class Post extends Sequelize.Model{
    static initiate(sequelize){
        Post.init({
            content: {
                type: Sequelize.TEXT,
                allowNull: false
            },
        },{
            sequelize,
            timestamps: true,
            underscored: false,
            modelName: 'Post',
            tableName: 'Posts',
            paranoid: false,
            charset: 'utf8mb4',
            collate: 'utf8mb4_general_ci',
        });
    }
    static associate(db) {
        db.Post.belongsTo(db.User, { foreignKey: 'userId', targetKey: 'userId' });
        db.Post.belongsToMany(db.HashTag, { through: 'PostHashTag' });
    }
}

module.exports = Post;
```

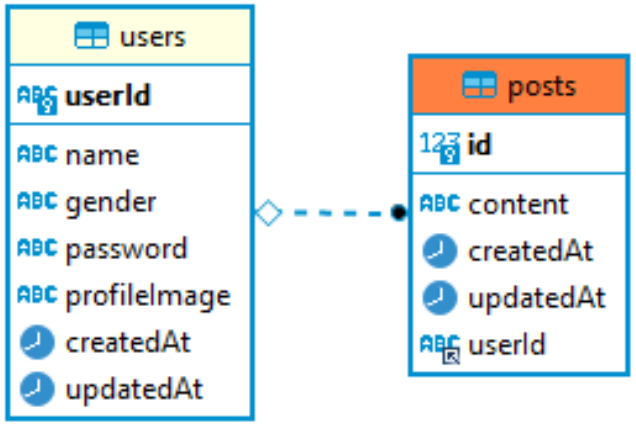
10. User, Post 모델 활성화하기

- index.js에 모델 연결
  - initiate으로 sequelize와 연결
  - associate로 관계 설정

```
'use strict';
const Sequelize = require('sequelize');
const User = require('./user');
const UserDetail = require('./userDetail');
const Post = require('./post');
const HashTag = require('./hashTag');
const PostHashTag = require('./postHashTag');
const process = require('process');
const env = process.env.NODE_ENV || 'development';

...

db.Sequelize = Sequelize;

db.User = User;
db.UserDetail = UserDetail;
db.Post = Post;
db.HashTag = HashTag;
db.PostHashTag = PostHashTag;

User.initiate(sequelize);
UserDetail.initiate(sequelize);
Post.initiate(sequelize);
HashTag.initiate(sequelize);
PostHashTag.initiate(sequelize);

User.associate(db);
UserDetail.associate(db);
Post.associate(db);

module.exports = db;
```

11. 관계 정의하기

- User 모델과 Post 모델 간의 관계를 정의
  - 1:N 관계 (사용자 한 명이 게시글 여러 개 작성)
  - 시퀄라이즈에서는 1:N 관계를 hasMany로 표현 (사용자.hasMany(댓글))
  - 반대의 입장에서는 belongsTo로 표현 (게시글.belongsTo(사용자))
  - belongsTo가 있는 테이블에 컬럼이 생김(게시글 테이블에 userId 컬럼)

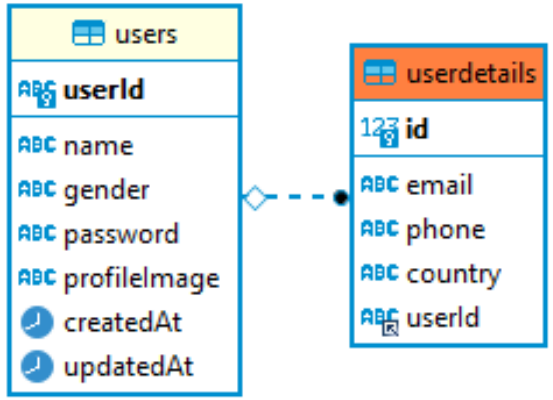



```
static associate(db) {
    db.User.hasMany(db.Post, { foreignKey: 'userId', sourceKey: 'userId' });
}

static associate(db) {
    db.Post.belongsTo(db.User, { foreignKey: 'userId', targetKey: 'userId' });
}
```

12. 1대1 관계

1대1 관계

- 예) 사용자 테이블과 사용자 정보 테이블



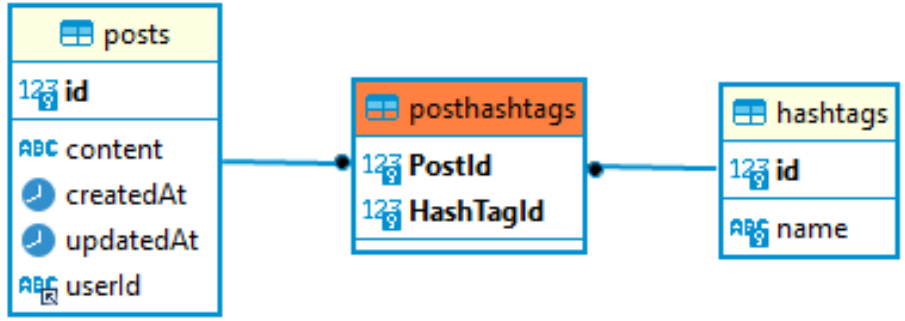

```
static associate(db) {
    db.User.hasOne(db.UserDetail, { foreignKey: 'userId', sourceKey: 'userId' });
}
```

```
static associate(db) {
        db.UserDetail.belongsTo(db.User, { foreignKey: 'userId', targetKey: 'userId' });
    }
```

13. N대M 관계

다대다 관계

- 예) 게시글과 해시태그 테이블
- 하나의 게시글이 여러 개의 해시태그를 가질 수 있고 하나의 해시태그가 여러 개의 게시글을 가질 수 있음



- DB 특성상 다대다 관계는 중간 테이블이 생김

```
static associate(db) {
    db.Post.belongsToMany(db.HashTag, { through: 'PostHashTag' });
}

static associate(db) {
    db.HashTag.belongsToMany(db.Post, { through: 'PostHashTag' });
}
```

14. 시퀄라이즈 쿼리 알아보기

- 윗 줄이 SQL문, 아랫 줄은 시퀄라이즈 쿼리(자바스크립트)

```
INSERT INTO `Users` (userid, name, gender, password, profile_image)
VALUES ('dlwlrma', '아이유', 'F', 'dlwlrma123', 'iu.png');

const users = User.create({
    userId: 'dlwlrma',
    name: '아이유',
    gender: 'F',
    password: 'dlwlrma123',
    profileImage: 'iu.png'
});


SELECT * FROM `users`;

const users = User.findAll({});


SELECT userId, name FROM `users`;

const users = User.findAll({
    attributes:['userId', 'name']
});
```

15. 시퀄라이즈 쿼리 알아보기

- 윗 줄이 SQL문, 아랫 줄은 시퀄라이즈 쿼리(자바스크립트)
  - 특수한 기능들인 경우 Sequelize.Op의 연산자 사용(gt, or 등)

```
const Op = require('sequelize').Op;
```

```
SELECT userId, name FROM `Users` WHERE name like '%아%';
```

```
User.findAll({
    attributes: ['userId', 'name'],
    where: {
        name: { [Op.like]: "%아%" }
    }
})
```

```
SELECT userId, name FROM `Users` WHERE gender='F' or name='아이유';
```

```
User.findAll({
    attributes: ['userId', 'name'],
    where: {
        [Op.or]: [{ gender: 'F' }, { name: '아이유' }]
    }
})
```

16. 시퀄라이즈 쿼리 알아보기

- 윗 줄이 SQL문, 아랫 줄은 시퀄라이즈 쿼리(자바스크립트)

```
SELECT id, name FROM `Users` ORDER BY id DESC;
const users = await User.findAll({
    attributes: ['id', 'name'],
    order: [['id', 'DESC']],
});

SELECT id, name FROM `Users` ORDER BY id DESC LIMIT 3;
const users = await User.findAll({
    attributes: ['id', 'name'],
    order: [['id', 'DESC']],
    limit: 3
});

SELECT id, name FROM `Users` ORDER BY id DESC LIMIT 3 OFFSET 1;
const users = await User.findAll({
    attributes: ['id', 'name'],
    order: [['id', 'DESC']],
    limit: 3,
    offset: 1
});
```

17. 시퀄라이즈 쿼리 알아보기

수정

```
UPDATE `Users` SET name = '아이유';
User.update({
    name: '아이유',
}, {
    where: {
        name: { [Op.like]: '%아%'}
    }
});
```

삭제

```
DELETE FROM `Users` WHERE id > 2;
User.destroy({
    where: {
        id: {[Op.gt]: 2}
    }
});
```

18. 관계 쿼리

- 결과값이 자바스크립트 객체임

```
const user = await User.findOne({});
console.log(user.nick); // 사용자 닉네임
```

- include로 JOIN 과 비슷한 기능 수행 가능(관계 있는 것 엮을 수 있음)

```
const user = await User.findOne({
  include: [{
    model: Comment,
  }]
});
console.log(user.Comments); // 사용자 댓글
```

- 다대다 모델은 다음과 같이 접근 가능

```
db.sequelize.models.PostHashtag
```

19. 관계 쿼리

- get+모델명으로 관계 있는 데이터 로딩 가능

```
const user = await User.findOne({});
const comments = await user.getComments();
console.log(comments); // 사용자 댓글
```

- as로 모델명 변경 가능

```
// 관계를 설정할 때 as로 등록
db.User.hasMany(db.Comment, { foreignKey: 'commenter', sourceKey: 'id', as: 'Answers'
});
// 쿼리할 때는
const user = await User.findOne({});
const comments = await user.getAnswers();
console.log(comments); // 사용자 댓글
```

20. 관계 쿼리

- include나 관계 쿼리 메서드에도 where나 attributes 사용 가능

```
const user = await User.findOne({
  include: [{
    model: Comment,
    where: {
      id: 1,
    },
    attributes: ['id'],
  }]
});
// 또는
const comments = await user.getComments({
  where: {
    id: 1,
  },
  attributes: ['id'],
});
```

- 생성 쿼리

```
const user = await User.findOne({});
const comment = await Comment.create();
await user.addComment(comment);
// 또는
await user.addComment(comment.id);
```

21. 관계 쿼리

- 여러 개를 추가할 때는 배열로 추가 가능

```
const user = await User.findOne({});
const comment1 = await Comment.create();
const comment2 = await Comment.create();
await user.addComment([comment1, comment2]);
```

- 덮어쓰기 set+모델명

- 삭제하기 remove+모델명

22. raw 쿼리

- 직접 SQL을 쓸 수 있음

```
const [result, metadata] = await sequelize.query('SELECT * from comments');
console.log(result);
```