# 8. Database

2023학년 2학기 웹응용프로그래밍

권 동 현

CSLAB

# Contents

# Introduction to MySQL

CSLAB

# What is Database?

- So far, we have been storing data in the server's memory.
  - When the server restarts, the data is lost, which necessitates a need for permanent storage.
- MySQL relational database
  - A database is a collection of related and non-redundant data
  - Database Management System (DBMS) is a system for managing databases
  - A Relational Database Management System (RDBMS) is a system for managing relational databases.
  - Data is stored on storage media such as the server's hard disk or SSD, allowing data to be continuously accessible regardless of server shutdown.
  - Multiple people can access the data simultaneously, and separate permissions can be assigned.

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

# MySQL –
# Create Database and Table

CSLAB

# Create Database

- In MySQL prompt
  - Create the "nodejs" database using **CREATE SCHEMA nodejs;**.
    - Set the character set to utf8mb4 for emoji support.
    - Collation determines how the character set is sorted.
  - Choose the newly created database with **USE nodejs;**.
  - Verify the list of tables in the database using **SHOW TABLES;**.

콘솔

```
mysql> CREATE SCHEMA `nodejs` DEFAULT CHARACTER SET utf8mb4 DEFAULT COLLATE utf8mb4_
general_ci;
Query OK, 1 row affected (0.01sec)
mysql> use nodejs;
Database changed
```

# Create Table

- You can create a table in MySQL using the following command:
  - **CREATE TABLE [database_name.table_name]**
- Table for storing user information: This table is used to store user information.

콘솔

```
mysql> CREATE TABLE nodejs.users (
    -> id INT NOT NULL AUTO_INCREMENT,
    -> name VARCHAR(20) NOT NULL,
    -> age INT UNSIGNED NOT NULL,
    -> married TINYINT NOT NULL,
    -> comment TEXT NULL,
    -> created_at DATETIME NOT NULL DEFAULT now(),
    -> PRIMARY KEY(id),
    -> UNIQUE INDEX name_UNIQUE (name ASC))
    -> COMMENT = '사용자 정보'
    -> ENGINE = InnoDB;
Query OK, 0 row affected (0.09 sec)
```

# Column and Row

- Information like age, marital status, gender, and similar details are represented as columns.
- The actual data that gets stored in the table is in rows.

Note ≡ 컬럼과 로우

그림 7-27 컬럼과 로우

# Column Options

- id INT **NOT NULL** **AUTO_INCREMENT**

  - INT: Integer data type (FLOAT and DOUBLE are for floating-point numbers)
  - VARCHAR: String data type with variable length (CHAR has fixed length)
  - TEXT: Used for storing long strings
  - DATETIME: Data type for storing date and time
  - TINYINT: Stores values from -128 to 127, but in this context, it's used to represent boolean values (1 or 0)

  - NOT NULL: Indicates that a column cannot contain empty values (NULL values are not allowed)
  - AUTO_INCREMENT: For numeric data types, automatically increments by 1 for each new row
  - UNSIGNED: Allows only non-negative (zero and positive) values
  - ZEROFILL: If the number has a fixed number of digits, it fills the empty spaces with zeros
  - DEFAULT now(): Sets the default value for a date column to the current time.

# Primary Key, Unique Index

- PRIMARY KEY(id)
  - The "id" column serves as a unique value within the table, allowing you to uniquely identify rows. This concept is similar to concepts like student IDs or social security numbers.

- UNIQUE INDEX name_UNIQUE (name ASC)
  - This option indicates that the specified column, "name" in this case, must have unique values. "name_UNIQUE" is the name of this option (you can choose any name you like). "ASC" signifies that the index stores values in ascending order (the opposite would be "DESC" for descending order).

```
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50) NOT NULL,
    student_email VARCHAR(100) UNIQUE,
    student_code INT
);
```

| student_id | student_name | student_email | student_code |
|---|---|---|---|
| 1 | Alice | alice@example.com | 101 |
| 2 | Bob | bob@example.com | 102 |
| 3 | Carol | NULL | 103 |

# Table Options

- COMMENT: Comments for Table
- ENGINE: Use InnoDB

# Check Table Creation

- DESC [Table Name]

콘솔

```
mysql> DESC users;
```

그림 7-27 DESC 명령어 결과

```
mysql> DESC users;
+------------+--------------+------+-----+-------------------+-------------------+
| Field      | Type         | Null | Key | Default           | Extra             |
+------------+--------------+------+-----+-------------------+-------------------+
| id         | int          | NO   | PRI | NULL              | auto_increment    |
| name       | varchar(20)  | NO   | UNI | NULL              |                   |
| age        | int unsigned | NO   |     | NULL              |                   |
| married    | tinyint(1)   | NO   |     | NULL              |                   |
| comment    | text         | YES  |     | NULL              |                   |
| created_at | datetime     | NO   |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED |
+------------+--------------+------+-----+-------------------+-------------------+
6 rows in set (0.00 sec)
```

- Delete Table: DROP TABLE [Table Name]

콘솔

```
mysql> DROP TABLE users;
```

# Create comments Table

콘솔

```
mysql> CREATE TABLE nodejs.comments (
    -> id INT NOT NULL AUTO_INCREMENT,
    -> commenter INT NOT NULL,
    -> comment VARCHAR(100) NOT NULL,
    -> created_at DATETIME NOT NULL DEFAULT now(),
    -> PRIMARY KEY(id),
    -> INDEX commenter_idx (commenter ASC),
    -> CONSTRAINT commenter
    -> FOREIGN KEY (commenter)
    -> REFERENCES nodejs.users (id)
    -> ON DELETE CASCADE
    -> ON UPDATE CASCADE)
    -> COMMENT = '댓글'
    -> ENGINE=InnoDB;
Query OK, 0 row affected (0.09 sec)
```

# Foreign key

- The comment table is related to the user table because users leave comments.
  - A foreign key is used to indicate this relationship between the two tables.
  - FOREIGN KEY (column_name) REFERENCES database.table_name (column)
  - e.g. FOREIGN KEY (commenter) REFERENCES nodejs.users (id)
  - a new column called "commenter" is created, and it references the "id" column of the "nodejs.users" table

  - ON DELETE CASCADE, ON UPDATE CASCADE
  - This means that when a row in the user table is deleted or updated, the associated rows in the comment table are also deleted or updated. These options are used to keep the data consistent. (Note that instead of CASCADE, options like SET NULL and NO ACTION can also be used.)

```
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(50)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id) ON UPDATE CASCADE
);
```

# Table list

- SHOW TABLES;

콘솔

```
mysql> SHOW TABLES;
+------------------+
| Tables_in_nodejs |
+------------------+
| comments         |
| users            |
+------------------+
2 rows in set (0.00 sec)
```

# MySQL – CRUD

# CRUD

- Create, Read, Update, Delete
  - Four main operations in database

그림 7-37 CRUD 작업

# Create

- INSERT INTO table (columns) VALUES (values)

콘솔

```
mysql> INSERT INTO nodejs.users (name, age, married, comment) VALUES ('zero', 24, 0, '자기소개1');
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO nodejs.users (name, age, married, comment) VALUES ('nero', 32, 1, '자기소개2');
Query OK, 1 row affected (0.02 sec)
```

콘솔

```
mysql> INSERT INTO nodejs.comments (commenter, comment) VALUES (1, '안녕하세요. zero의 댓글입니다');
Query OK, 1 row affected (0.02 sec)
```

# Read

- SELECT column FROM table
  - SELECT * means selecting all columns
  - It is also possible to select only the columns separately.

콘솔

```
mysql> SELECT * FROM nodejs.users;
+----+------+-----+---------+---------+---------------------+
| id | name | age | married | comment | created_at          |
+----+------+-----+---------+---------+---------------------+
|  1 | zero |  24 |       0 | 자기소개1    | 2017-10-25 07:06:33 |
|  2 | nero |  32 |       1 | 자기소개2    | 2017-10-25 09:25:40 |
+----+------+-----+---------+---------+---------------------+
2 rows in set (0.00 sec)
```

콘솔

```
mysql> SELECT name, married FROM nodejs.users;
+------+---------+
| name | married |
+------+---------+
| zero |       0 |
| nero |       1 |
+------+---------+
2 rows in set (0.00 sec)
```

# Read options

- Conditions can be applied using "WHERE" to make selections based on specific criteria
  - AND: Multiple conditions can be combined using "AND" to find records that satisfy all of them simultaneously.
  - OR: Multiple conditions can be combined using "OR" to find records that satisfy at least one of them.

콘솔

```
mysql> SELECT name, age FROM nodejs.users WHERE married = 1 AND age > 30;
+------+-----+
| name | age |
+------+-----+
| nero |  32 |
+------+-----+
1 row in set (0.00 sec)
```

콘솔

```
mysql> SELECT id, name FROM nodejs.users WHERE married = 0 OR age > 30;
+----+------+
| id | name |
+----+------+
|  1 | zero |
|  2 | nero |
+----+------+
2 rows in set (0.01 sec)
```

# Read options

- You can order the results by a specific column using "ORDER BY."
  - "DESC" stands for descending order (highest to lowest), and "ASC" stands for ascending order (lowest to highest).

콘솔

```
mysql> SELECT id, name FROM nodejs.users ORDER BY age DESC;
+----+------+
| id | name |
+----+------+
|  2 | nero |
|  1 | zero |
+----+------+
2 rows in set (0.01 sec)
```

# Read options

- You can use "LIMIT" to limit the number of rows returned in a query

콘솔

```
mysql> SELECT id, name FROM nodejs.users ORDER BY age DESC LIMIT 1;
+----+------+
| id | name |
+----+------+
|  2 | nero |
+----+------+
1 row in set (0.00 sec)
```

- "OFFSET" allows you to skip a certain number of rows, starting from the beginning
  - For example, "OFFSET 2" means you skip the first two rows and start from the third row in the result set.

콘솔

```
mysql> SELECT id, name FROM nodejs.users ORDER BY age DESC LIMIT 1 OFFSET 1;
+----+------+
| id | name |
+----+------+
|  1 | zero |
+----+------+
1 row in set (0.00 sec)
```

# Update

- To update data in a database
  - UPDATE table SET column=new_value WHERE conditions

콘솔

```
mysql> UPDATE nodejs.users SET comment = '바꿀 내용' WHERE id = 2;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

# 8. Delete

- To delete data in a database
  - DELETE FROM table WHERE conditions

콘솔

```
mysql> DELETE FROM nodejs.users WHERE id = 2;
Query OK, 1 row affected (0.00 sec)
```

# MySQL – sequelize

# Use sequelize CLI

- Install sequelize-cli to use the sequelize command
  - mysql2 is a driver, not a MySQL DB (connects Node.js and MySQL)

콘솔

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

- Create a sequel structure with 'npx sequelize init'

콘솔

```
$ npx sequelize init
Sequelize CLI [Node: 18.0.0, CLI: 6.4.1, ORM: 6.19.0]
Created "config\config.json"
Successfully created models folder at ...
Successfully created migrations folder at ...
Successfully created seeders folder at ...
```

# Modify models/index.js

- Modify it as follows:
  - require(../config/config) Loading configuration
  - Can connect to DB with new Sequelize (options...)

models/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);
db.sequelize = sequelize;

module.exports = db;
```

# Connect MySQL

- Write app.js
  - connect MySQL with sequelize.sync

app.js

```
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 || 3001);
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'));
app.use(express.static(path.join(__dirname, 'public')));
app.use(express.json());
app.use(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 !== 'production' ? err : {};
  res.status(err.status || 500);
  res.render('error');
});

app.listen(app.get('port'), () => {
  console.log(app.get('port'), '번 포트에서 대기 중');
});
```

# Configure config.json

- Enter DB connection information

config/config.json

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

# Test Connection

- Run npm start and if SELECT 1+1 AS RESULT appears, the connection is successful.

콘솔

```
$ npm start
> learn-sequelize@0.0.1 start
> nodemon app

[nodemon] 2.0.16
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
3001 번 포트에서 대기 중
Executing (default): SELECT 1+1 AS result
데이터베이스 연결 성공
```

# Create Models

- Create squelize model corresponding to the table
  - static initiate / static associate

models/user.js

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

class User extends Sequelize.Model {
  static initiate(sequelize) {
    User.init({
      name: {
        type: Sequelize.STRING(20),
        allowNull: false,
        unique: true,
      },
      age: {
        type: Sequelize.INTEGER.UNSIGNED,
        allowNull: false,
      },
      married: {
        type: Sequelize.BOOLEAN,
        allowNull: false,
      },
      comment: {
        type: Sequelize.TEXT,
        allowNull: true,
      },
      created_at: {
        type: Sequelize.DATE,
        allowNull: false,
        defaultValue: Sequelize.NOW,
      },
    }, {
      sequelize,
      timestamps: false,
      underscored: false,
      modelName: 'User',
      tableName: 'users'.
      paranoid: false,
      charset: 'utf8',
      collate: 'utf8_general_ci',
    });
  }

  static associate(db) {}
};

module.exports = User;
```

# Model options

- The Sequelize data types are slightly different from MySQL data types.

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 |

- Table options
  - If timestamps is set to true, it automatically creates createdAt (creation time) and updatedAt (modification time) columns.
    - In the example, the created_at column was manually created, so it's set to false.
  - The paranoid option, when set to true, creates a deletedAt (deletion time) column. It keeps a record of deletions without completely removing the row.
  - The underscored option changes the column naming convention from camel case to snake case.
  - modelName is the model name, and the tableName option sets the table name.
  - charset and collate are necessary for Korean settings (use utf8mb4 for emoji support).

# Create comment model

- comment.js

models/comment.js

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

class Comment extends Sequelize.Model {
  static initiate(sequelize) {
    Comment.init({
      comment: {
        type: Sequelize.STRING(100),
        allowNull: false,
      },
      created_at: {
        type: Sequelize.DATE,
        allowNull: true,
        defaultValue: Sequelize.NOW,
      },
    }, {
      sequelize,
      timestamps: false,
      modelName: 'Comment',
      tableName: 'comments',
      paranoid: false,
      charset: 'utf8mb4',
      collate: 'utf8mb4_general_ci',
    });
  }

  static associate(db) {
    db.Comment.belongsTo(db.User, { foreignKey: 'commenter', targetKey: 'id' });
  }
};

module.exports = Comment;
```

# Activate Models

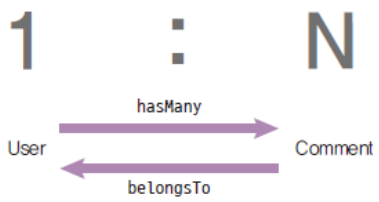
- Connect models in index.js.
  - Use initiate to connect Sequelize.
  - Use associate to establish relationships.

models/index.js

```
const Sequelize = require('sequelize');
const User = require('./user');
const Comment = require('./comment');

...
db.sequelize = sequelize;

db.User = User;
db.Comment = Comment;

User.initiate(sequelize);
Comment.initiate(sequelize);

User.associate(db);
Comment.associate(db);

module.exports = db;
```

# Define Relationship

- Define the relationship between the users model and the comments model.
  - It's a 1:N relationship (one user writes multiple comments).
  - In Sequelize, a 1:N relationship is represented using hasMany (User.hasMany(Comment)).
  - From the opposite perspective, it's belongsTo (Comment.belongsTo(User)).
  - The belongsTo association adds a column to the table it's in (the comments table in this case, with a commenter column).



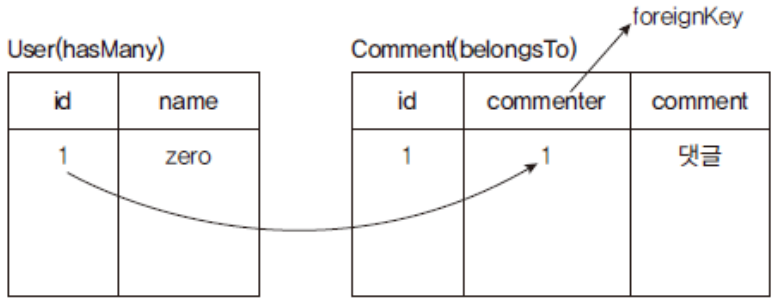

모델 각각의 static associate 메서드에 넣습니다.

**models/user.js**

```
...
  static associate(db) {
    db.User.hasMany(db.Comment, { foreignKey: 'commenter', sourceKey: 'id' });
  }
};
```

**models/comment.js**

```
...
  static associate(db) {
    db.Comment.belongsTo(db.User, { foreignKey: 'commenter', targetKey: 'id' });
  }
};
```



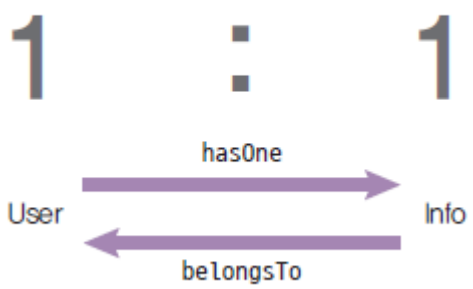

commenter는 foreignKey
User의 id는 hasMany의 sourceKey이자
belongsTo의 targetKey

# 1:1 Relationship

- 1:1 relationship
  - Example) User table and user information table

```
db.User.hasOne(db.Info, { foreignKey: 'UserId', sourceKey: 'id' });
db.Info.belongsTo(db.User, { foreignKey: 'UserId', targetKey: 'id' });
```

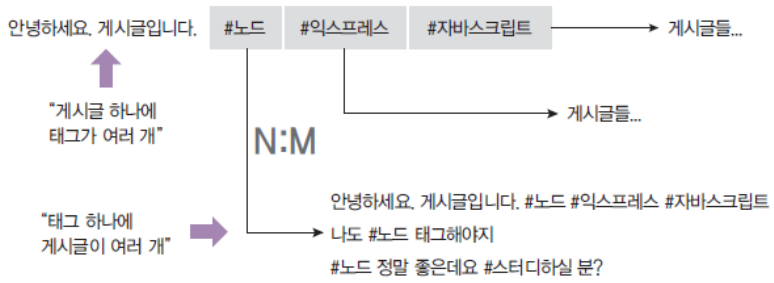
그림 7-56 1:1 관계



# N:M Relationship

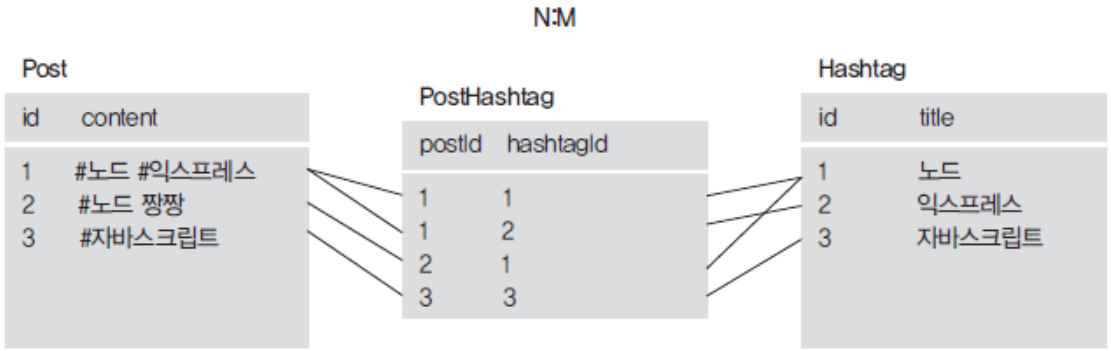
- many-to-many relationship
  - Example) Post and hashtag table
  - One post can have multiple hashtags and one hashtag can have multiple posts.
  - Due to the nature of DB, many-to-many relationships create intermediate tables.

그림 7-57 N:M 관계




```
db.Post.belongsToMany(db.Hashtag, { through: 'PostHashtag' });
db.Hashtag.belongsToMany(db.Post, { through: 'PostHashtag' });
```

그림 7-58 N:M 관계 테이블



# Squelize Queries

- The top line is an SQL statement, the bottom line is a sequelize query (JavaScript).

```
INSERT INTO nodejs.users (name, age, married, comment) VALUES ('zero', 24, 0,
'자기소개1');
const { User } = require('../models');
User.create({
  name: 'zero',
  age: 24,
  married: false,
  comment: '자기소개1',
});

SELECT * FROM nodejs.users;
User.findAll({});

SELECT name, married FROM nodejs.users;
User.findAll({
  attributes: ['name', 'married'],
});
```

# Squelize Queries

- For special functions, use Sequelize.Op operators (gt, or, etc.)

```
SELECT name, age FROM nodejs.users WHERE married = 1 AND age > 30;
const { Op } = require('sequelize');
const { User } = require('../models');
User.findAll({
  attributes: ['name', 'age'],
  where: {
    married: 1,
    age: { [Op.gt]: 30 },
  },
});
```

```
SELECT id, name FROM users WHERE married = 0 OR age > 30;
const { Op } = require('sequelize');
const { User } = require('../models');
User.findAll({
  attributes: ['id', 'name'],
  where: {
    [Op.or]: [{ married: 0 }, { age: { [Op.gt]: 30 } }],
  },
});
```

# Squelize Queries

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

```
SELECT id, name FROM users ORDER BY age DESC LIMIT 1;
User.findAll({
  attributes: ['id', 'name'],
  order: [['age', 'DESC']],
  limit: 1,
});
```

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

# Squelize Queries

- Update

```
UPDATE nodejs.users SET comment = '바꿀 내용' WHERE id = 2;
User.update({
  comment: '바꿀 내용',
}, {
  where: { id: 2 },
});
```

- Delete

```
DELETE FROM nodejs.users WHERE id = 2;
User.destory({
  where: { id: 2 },
});
```

# Relationship Query

- The result is a JavaScript object.

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

- Functions similar to JOIN can be performed with include (related things can be linked).

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

- The many-to-many model can be accessed as follows:

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

# Relationship Query

- You can load related data using "get" followed by the model name
  - For example, besides "getComments" for retrieving, there are also methods like "setComments" for updating, "addComment" for creating one, "addComments" for creating multiple, and "removeComments" for deleting.

- You can use "as" to change the model name as well.

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

```
// 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
```

# Relationship Query

- where or attributes in include or relational query methods

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

- Add data

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

# Relationship Query

- When adding multiple items, they can be added as an array.

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

- Modification is set+model name, deletion is remove+model name.

# Raw query

- Can use SQL commands directly

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

# MySQL Example

- https://github.com/zerocho/nodejs-book/tree/master/ch7/7.6/learn-sequelize

- Focus on server code rather than front code
  - The front code only focuses on AJAX requests that send requests to the server.

- routes/index.js

routes/index.js

```
const express = require('express');
const User = require('../models/user');

const router = express.Router();

router.get('/', async (req, res, next) => {
  try {
    const users = await User.findAll();
    res.render('sequelize', { users });
  } catch (err) {
    console.error(err);
    next(err);
  }
});

module.exports = router;
```

```
<tbody>
    {% for user in users %}
    <tr>
      <td>{{user.id}}</td>
      <td>{{user.name}}</td>
      <td>{{user.age}}</td>
      <td>{{ '기혼' if user.married else '미혼'}}</td>
    </tr>
    {% endfor %}
  </tbody>
```

# MySQL Example

- users router
  - handle get, post, delete, patch requests
  - Data response in JSON format
  - Similar with comments router

routes/users.js

```
const express = require('express');
const User = require('../models/user');
const Comment = require('../models/comment');

const router = express.Router();

router.route('/')
  .get(async (req, res, next) => {
    try {
      const users = await User.findAll();
      res.json(users);
    } catch (err) {
      console.error(err);
      next(err);
    }
  })
  .post(async (req, res, next) => {
    try {
      const user = await User.create({
        name: req.body.name,
        age: req.body.age,
        married: req.body.married,
      });
      console.log(user);
      res.status(201).json(user);
    } catch (err) {
      console.error(err);
      next(err);
    }
  });

router.get('/:id/comments', async (req, res, next) => {
  try {
    const comments = await Comment.findAll({
      include: {
        model: User,
        where: { id: req.params.id },
      },
    });
    console.log(comments);
    res.json(comments);
  } catch (err) {
    console.error(err);
    next(err);
  }
});

module.exports = router;
```

# MySQL Example

- comments router

routes/comments.js

```
const express = require('express');
const { User, Comment } = require('../models');

const router = express.Router();

router.post('/', async (req, res, next) => {
  try {
    const comment = await Comment.create({
      commenter: req.body.id,
      comment: req.body.comment,
    });
    console.log(comment);
    res.status(201).json(comment);
  } catch (err) {
    console.error(err);
    next(err);
  }
});
```

```
router.route('/:id')
  .patch(async (req, res, next) => {
    try {
      const result = await Comment.update({
        comment: req.body.comment,
      }, {
        where: { id: req.params.id },
      });
      res.json(result);
    } catch (err) {
      console.error(err);
      next(err);
    }
  })
  .delete(async (req, res, next) => {
    try {
      const result = await Comment.destroy({ where: { id: req.params.id } });
      res.json(result);
    } catch (err) {
      console.error(err);
      next(err);
    }
  });

module.exports = router;
```

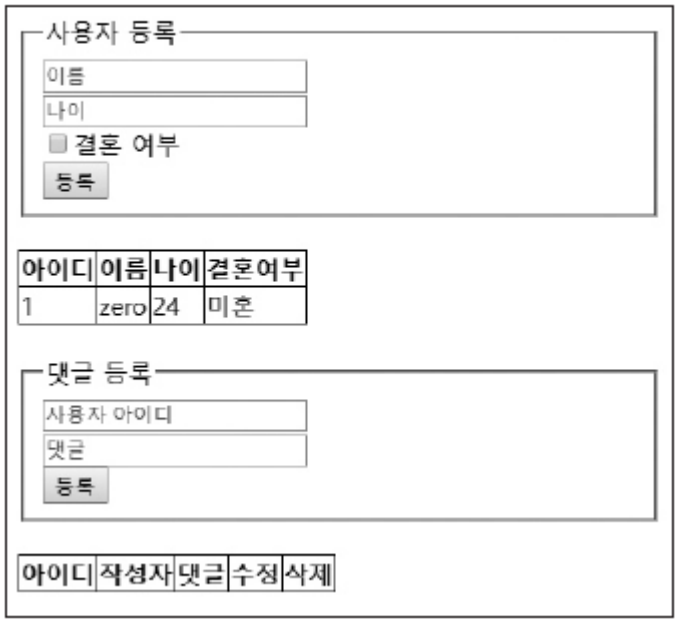
# MySQL Example

- Start the server with npm start
  - When connected to localhost:3001, the SQL statement executed by Sequelize is displayed in the console.

**콘솔**

```
Executing (default): SELECT `id`, `name`, `age`, `married`, `comment`, `created_at` FROM
`users` AS `users`;
// 이하 생략
```

그림 7-59 접속 화면



# MySQL Example

- Try registering/editing/deleting a post

그림 7-61 사용자 이름 클릭 시 화면

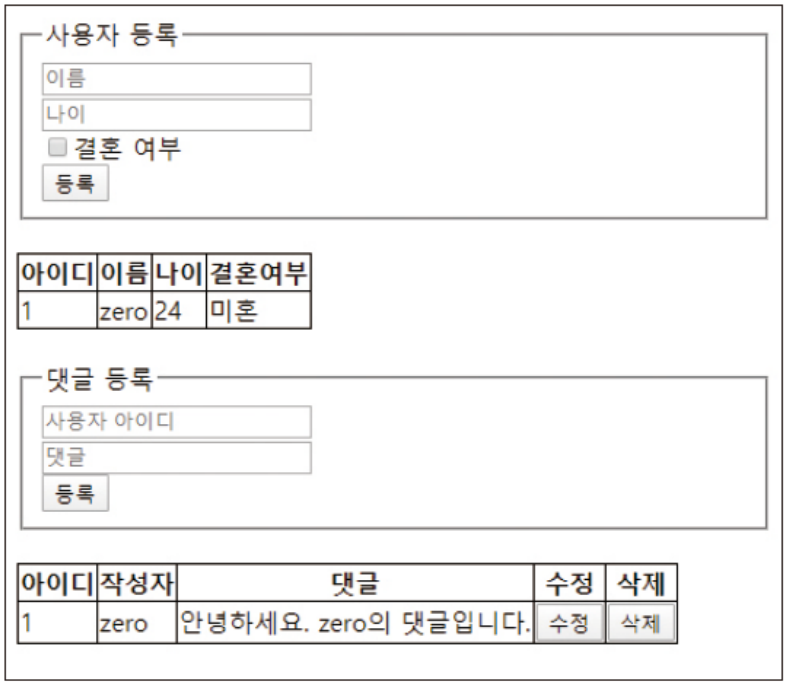
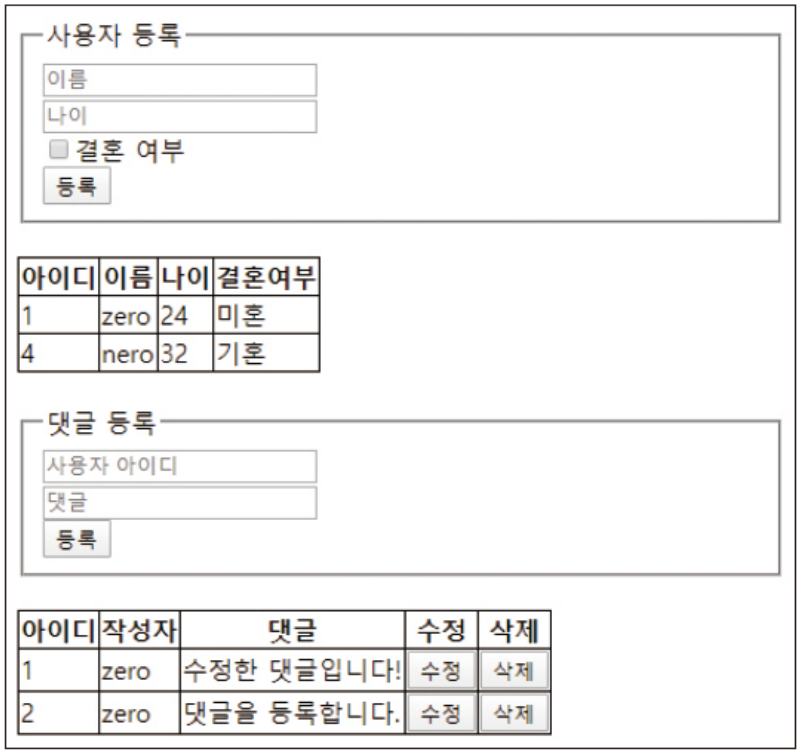



그림 7-62 nero 사용자 등록과 zero 댓글 작성 후 화면



# Introduction to MongoDB

CSLAB

# NoSQL

- A different type of data from SQL databases such as MySQL
  - Using mongoDB, a leader in NoSQL

그림 8-1 몽고디비 로고


표 8-1 SQL과 NoSQL의 비교

| SQL(MySQL) | NoSQL(몽고디비) |
|---|---|
| 규칙에 맞는 데이터 입력 | 자유로운 데이터 입력 |
| 테이블 간 JOIN 지원 | 컬렉션 간 JOIN 미지원 |
| 안정성, 일관성 | 확장성, 가용성 |
| 용어(테이블, 로우, 컬럼) | 용어(컬렉션, 다큐먼트, 필드) |

- JOIN: Function to combine data between tables with relationships (can be imitated with MongoDB aggregate)
- MongoDB is recommended for big data, messaging, session management, etc. (unstructured data).

# MongoDB –
# Create Database and Collection

CSLAB

# Create Database

- use [database]

콘솔

```
> use nodejs
switched to db nodejs
```

- show 윱

콘솔

```
> show dbs
admin 0.000GB
config 0.000GB
local 0.000GB
```

- db

콘솔

```
> db
nodejs
```

# Create Collection

- No need to create it separately
  - The moment you insert a document, a collection is automatically created.
  - There are also commands to create your own.

콘솔

```
> db.createCollection('users')
{ "ok" : 1 }
> db.createCollection('comments')
{ "ok" : 1 }
```

- Check current collections with show collections

콘솔

```
> show collections
comments
users
```

# MongoDB – CRUD

# Create

- MongoDB does not require defining columns.
  - The advantage of being free, the disadvantage of not knowing what will happen
  - Follows JavaScript data type (there are differences)
  - ObjectId: MongoDB data type that serves as a unique ID
  - Save with insertOne method

콘솔

```
$ mongosh
test> use nodejs;
switched to db nodejs
nodejs> db.users.insertOne({ name: 'zero', age: 24, married: false, comment: '안녕하세요.
➥ 간단히 몽고디비 사용 방법에 대해 알아봅시다.', createdAt: new Date() });
{
  acknowledged: true,
  insertedId: ObjectId("5a1687007af03c3700826f70")
}
nodejs> db.users.insertOne({ name: 'nero', age: 32, married: true, comment: '안녕하세요.
➥ zero 친구 nero입니다.', createdAt: new Date() });
{
  acknowledged: true,
  insertedId: ObjectId("62fba0deb068d84d69d7c740")
}
```

# Create (Relationship)

- There are no restrictions that force relationships between collections, so you can directly enter the ObjectId to connect them.
  - Find the user's ObjectId and place it in the comments collection.

콘솔

```
nodejs> db.users.find({ name: 'zero' }, { _id: 1 })
[ { "_id" : ObjectId("5a1687007af03c3700826f70") } ]
```

콘솔

```
nodejs> db.comments.insertOne({ commenter: ObjectId('5a1687007af03c3700826f70'),
➥ comment: '안녕하세요. zero의 댓글입니다.', createdAt: new Date() });
{
  acknowledged: true,
  insertedId: ObjectId("62fba1b6b068d84d69d7c741")
}
```

# Read

- use find()

콘솔

```
nodejs> db.users.find({});
[
  { "_id" : ObjectId("5a1687007af03c3700826f70"), "name" : "zero", "age" : 24, "mar-
ried" : false, "comment" : "안녕하세요. 간단히 몽고디비 사용 방법을 알아봅시다.", "createdAt"
: ISODate("2022-04-30T05:00:00Z") },
  { "_id" : ObjectId("5a16877b7af03c3700826f71"), "name" : "nero", "age" : 32, "mar-
ried" : true, "comment" : "안녕하세요. zero 친구 nero입니다.", "createdAt" : ISODate("2017-
11-23T01:00:00Z") }
]
nodejs> db.comments.find({})
[ { "_id" : ObjectId("5a1687e67af03c3700826f73"), "commenter" : ObjectId("5a1687007af-
03c3700826f70"), "comment" : "안녕하세요. zero의 댓글입니다.", "createdAt" : ISODate("2022-
04-30T05:30:00Z") } ]
```

# Read (conditions)

- The second argument allows you to select the field to query (1 to add, 0 to exclude)

콘솔

```
nodejs> db.users.find({}, { _id: 0, name: 1, married: 1 });
[
  { "name" : "zero", "married" : false },
  { "name" : "nero", "married" : true }
]
```

- Search conditions can be entered as the first argument
  - Use conditional operators like $gt or $or

콘솔

```
> db.users.find({ age: { $gt: 30 }, married: true }, { _id: 0, name: 1, age: 1 });
{ "name" : "nero", "age" : 32 }
```

콘솔

```
> db.users.find({ $or: [{ age: { $gt: 30 } }, { married: false }] }, { _id: 0, name: 1,
age: 1 });
{ "name" : "zero", "age" : 24 }
{ "name" : "nero", "age" : 32 }
```

# Read (Conditions)

- Sorting is done using the sort method.
  - -1 is in descending order, 1 is in ascending order

콘솔

```
> db.users.find({}, { _id: 0, name: 1, age: 1 }).sort({ age: -1 })
{ "name" : "nero", "age" : 32 }
{ "name" : "zero", "age" : 24 }
```

- Limit the number of documents to be viewed using the limit method

콘솔

```
> db.users.find({}, { _id: 0, name: 1, age: 1 }).sort({ age: -1 }).limit(1)
{ "name" : "nero", "age" : 32 }
```

- Provide the number of documents to skip with the skip method

콘솔

```
> db.users.find({}, { _id: 0, name: 1, age: 1 }).sort({ age: -1 }).limit(1).skip(1)
{ "name" : "zero", "age" : 24 }
```

# 6. Update

- Query with update method
  - Provide the modification target as the first argument and the modification content as the second argument.
  - Be careful because if you do not add $set, the entire document will be replaced.

콘솔

```
nodejs> db.users.updateOne({ name: 'nero' }, { $set: { comment: '안녕하세요. 이 필드를
➥ 바꿔보겠습니다!' } });
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 0,
  upsertedCount: 0
}
```

# 7. Delete

- Query with deleteOne method
  - Provide target condition to delete as first argument
  - On success, the deleted count is returned.

콘솔

```
nodejs> db.users.deleteOne({ name: 'nero' })
{ acknowledged: true, deletedCount: 1 }
```

# MongoDB – Mongoose

# Mongoose ODM

- A library that helps you work with MongoDB easily
  - ODM: Object Document Mapping: Mapping objects and documents (1:1 pairing)
  - Mongoose complements the inconvenient functions that MongoDB does not have.
  - Table-like function, JOIN function added
- https://github.com/zerocho/nodejs-book/tree/master/ch8/8.6/learn-mongoose

package.json

```
{
  "name": "learn-mongoose",
  "version": "0.0.1",
  "description": "몽구스를 배우자",
  "main": "app.js",
  "scripts": {
    "start": "nodemon app"
  },
  "author": "ZeroCho",
  "license": "MIT"
}
```

콘솔

```
$ npm i express morgan nunjucks mongoose
$ npm i -D nodemon
```

# Connect to MongoDB

- Authentication is from the admin database, service is from the dbName database.

mongodb://[username:password@]host[:port][/[database][?options]]

schemas/index.js

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

const connect = () => {

  if (process.env.NODE_ENV !== 'production') {
    mongoose.set('debug', true);
  }

  mongoose.connect('mongodb://root:nodejsbook@localhost:27017/admin', {
    dbName: 'nodejs',
    useNewUrlParser: true,
  }, (error) => {
    if (error) {
      console.log('몽고디비 연결 에러', error);
    } else {
      console.log('몽고디비 연결 성공');
    }
  });
};
mongoose.connection.on('error', (error) => {
  console.error('몽고디비 연결 에러', error);
});
mongoose.connection.on('disconnected', () => {
  console.error('몽고디비 연결이 끊겼습니다. 연결을 재시도합니다.');
  connect();
});

module.exports = connect;
```

# Connect app.js

- App.js

app.js

```
const express = require('express');
const path = require('path');
const morgan = require('morgan');
const nunjucks = require('nunjucks');

const connect = require('./schemas');

const app = express();
app.set('port', process.env.PORT || 3002);
app.set('view engine', 'html');
nunjucks.configure('views', {
  express: app,
  watch: true,
});
connect();

app.use(morgan('dev'));
app.use(express.static(path.join(__dirname, 'public')));
app.use(express.json());
app.use(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 !== 'production' ? err : {};
  res.status(err.status || 500);
  res.render('error');
});

app.listen(app.get('port'), () => {
  console.log(app.get('port'), '번 포트에서 대기 중');
});
```

# Define schemas

- Written in schemas folder
  - Forces only certain data to be entered, like a MySQL table.
  - type is the data type, require is required, default is the default value, and unique is unique.

schemas/user.js

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

const { Schema } = mongoose;
const userSchema = new Schema({
  name: {
    type: String,
    required: true,
    unique: true,
  },
  age: {
    type: Number,
    required: true,
  },
  married: {
    type: Boolean,
    required: true,
  },
  comment: String,
  createdAt: {
    type: Date,
    default: Date.now,
  },
});

module.exports = mongoose.model('User', userSchema);
```

schemas/comment.js

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

const { Schema } = mongoose;
const { Types: { ObjectId } } = Schema;
const commentSchema = new Schema({
  commenter: {
    type: ObjectId,
    required: true,
    ref: 'User',
  },
  comment: {
    type: String,
    required: true,
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
});

module.exports = mongoose.model('Comment', commentSchema);
```

# Router

routes/index.js

```
const express = require('express');
const User = require('../schemas/user');

const router = express.Router();

router.get('/', async (req, res, next) => {
  try {
    const users = await User.find({});
    res.render('mongoose', { users });
  } catch (err) {
    console.error(err);
    next(err);
  }
});

module.exports = router;
```

# User Router

routes/users.js

```
const express = require('express');
const User = require('../schemas/user');
const Comment = require('../schemas/comment');

const router = express.Router();

router.route('/')
  .get(async (req, res, next) => {
    try {
      const users = await User.find({});
      res.json(users);
    } catch (err) {
      console.error(err);
      next(err);
    }
  })
  .post(async (req, res, next) => {
    try {
      const user = await User.create({
        name: req.body.name,
        age: req.body.age,
        married: req.body.married,
      });
      console.log(user);
      res.status(201).json(user);
    } catch (err) {
      console.error(err);
      next(err);
    }
  });
```

```
router.get('/:id/comments', async (req, res, next) => {
  try {
    const comments = await Comment.find({ commenter: req.params.id })
      .populate('commenter');
    console.log(comments);
    res.json(comments);
  } catch (err) {
    console.error(err);
    next(err);
  }
});

module.exports = router;
```

# Comment Router

routes/comments.js

```
const express = require('express');
const Comment = require('../schemas/comment');

const router = express.Router();

router.post('/', async (req, res, next) => {
  try {
    const comment = await Comment.create({
      commenter: req.body.id,
      comment: req.body.comment,
    });
    console.log(comment);
    const result = await Comment.populate(comment, { path: 'commenter' });
    res.status(201).json(result);
  } catch (err) {
    console.error(err);
    next(err);
  }
});

router.route('/:id')
  .patch(async (req, res, next) => {
    try {
      const result = await Comment.update({
        _id: req.params.id,
      }, {
        comment: req.body.comment,
      });
      res.json(result);
    } catch (err) {
      console.error(err);
      next(err);
    }
  })
  .delete(async (req, res, next) => {
    try {
      const result = await Comment.remove({ _id: req.params.id });
      res.json(result);
    } catch (err) {
      console.error(err);
      next(err);
    }
  });

module.exports = router;
```

# Connect Router

- app.js

app.js

```
const connect = require('./schemas');
const indexRouter = require('./routes');
const usersRouter = require('./routes/users');
const commentsRouter = require('./routes/comments');

const app = express();
...
app.use(express.urlencoded({ extended: false }));

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

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

# Start Server

- After npm start, connect to localhost:3002
  - Check MongoDB queries recorded in console

콘솔

```
$ npm start
> learn-mongoose@0.0.1 start
> nodemon app

[nodemon] 2.0.16
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
3002 번 포트에서 대기 중
몽고디비 연결 성공
Mongoose: users.createIndex({ name: 1 }, { unique: true, background: true })
```

# Test Server

- Try registering/editing/deleting users/comments

그림 8-35 몽구스 서버 화면