Web and Mobile Application Development

Interactions Between Node.js & Relational Databases

Material created by:
David Augenblick, Bill Mongan, Dan Ziegler, Samantha Bewley, and Matt Burlick

What are Relational Databases?

- Relational databases provide a set structure to store data.
- They also allow us to **relate** data in different sets (called tables) together without duplication.
- This allows us to keep an accurate data set that's lean yet robust.
- Relational databases contain the following characteristics:
    1. Tables are designed to minimize data redundancy

Example – If you want to store information about students and their course grades:

- Student "static" information (name, ID, age, major, etc.) could be stored in one table, while "transactional data (ID, course#, grade, term taken, etc.) could reside in a separate table.

- If both were placed in a single table, a student's information would need to be part of every course record, resulting in a significant amount of redundancy.

What are Relational Databases?

- Relational databases contain the following characteristics:

2. Line items (rows) within a given table must be unique.
    - One or more fields (columns) must be designated as the “key” for that table. Every table must contain a key. The value(s) in the “key” fields uniquely define the entire row.
3. By having a field in 2 different tables that contains the same information (eg. Student id), it becomes possible to merge (“join”) fields of 2 different tables.
    - That field is usually the designated key for one of the tables, while in the other table it is referred to as a “foreign” key.

More on Database and Table Structures

- As mentioned, tables should have a unique **key** column that differentiates the row from any other row
  - Could be auto incremented (each new row is an id +1 of the previous row)
  - Should be unique (`userID`)
- Each column should be named something descriptive
  - Your table will be used (if a longstanding app) for years to come – make sure that the names you use today will be used by the new hire in the future!
- You'll need to choose a data type for each column, such as these general categories:
  - `INT, FLOAT, DOUBLE`
  - `DATE, DATETIME, TIME, TIMESTAMP, YEAR`
  - `CHAR, VARCHAR, BLOB, TEXT, ENUM`

What is MySQL?

- MySQL is an open source SQL (Relational) database server / management system (RDBMS) than can run on Linux (and Win32).
- Other RDBMSs include Oracle, MS SQL Server, MS Access, DB2 and many more
- MySQL contains administrative features to coordinate access to the database along with tools to create, populate, modify and delete tables within the database. SQL (Structured Query Language) is the vehicle by which these operations are accomplished (more about SQL on the following slide).

What is SQL

- SQL (Structured Query Language) is the tool used to communicate with relational databases
- It is the universal standard language for use in relational database management systems (RDBMS)
  - Although there could exist some minor differences between RDBMSs
- We use it to grant access permissions, create and modify databases and their tables, and to send queries to the database.

Database Creation

- Creating a MySQL database via SQL
  - Install MySQL and create database
  - Creating tables
  - Populating tables
  - Operations on tables

Database Creation

- You may want to create a MySQL database locally.
- To do so first download the *MySQL Community Server* for your operating system.
- During installation
  - You can choose Developer Default, but for our purposes Server only will be suffice.
  - Choose your MySQL Root Password and add a MySQL user. You'll need these for connecting to the database.
- Once installed you should have a program called *MySQL Command Line Client*
  - Here you can do mysql commands like:

```
CREATE DATABASE <dbname>;
SHOW DATABASES;
USE <dbname>;
```

  - Don't forget the semicolons!

Creating a Table

- We need something to put our data into!
- So, creating a table is always the first step in your database creating endeavors.
- To the right is a basic script to create a table. Some key points…
- Each column is listed first followed by its data type.
  `VARCHAR` is a type of text (and its limit is 20 characters)
- Running `describe <tablename>;` will show you the current table structure.

```
CREATE TABLE pet (
    id INT key,
    name VARCHAR(20),
    owner VARCHAR(20),
    species VARCHAR(20),
    sex CHAR(1),
    birth DATE
);
```

-- Once you do that, you can run the following query and you'll be able to see your structure.

```
DESCRIBE pet;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| id      | int         | YES  | YES | NULL    |       |
| name    | varchar(20) | YES  |     | NULL    |       |
| owner   | varchar(20) | YES  |     | NULL    |       |
| species | varchar(20) | YES  |     | NULL    |       |
| sex     | char(1)     | YES  |     | NULL    |       |
| birth   | date        | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

Creating a Table

- If the table has a naturally unique column, we can just use it as the key

```
id INT key
```

- Otherwise we should create one for the table and have it auto-increment:

```
id INT key AUTO_INCREMENT
```

```
CREATE TABLE pet (
    id INT key AUTO_INCREMENT,
    name VARCHAR(20),
    owner VARCHAR(20),
    species VARCHAR(20),
    sex CHAR(1),
    birth DATE
);
```

Inserting Rows

- Now that we have a table, we'll want to add a brand new row to our table.
- Let's add our first pet!

```
INSERT INTO pet
( id, name, owner, species, sex, birth )
VALUES
( 1, 'Maxie', 'Judy', 'Cat', 'F', '1987-2-14' );

-- you can insert multiple rows today

INSERT INTO pet
( id, name, owner, species, sex, birth )
VALUES
( 1, 'Maxie', 'Judy', 'Cat', 'F', '1987-2-14' ),
( 2, 'Shelly', 'Judy', 'Cat', 'F', '2004-1-05' ),
( 3, 'Buddy', 'Dan', 'Hamster', 'M', '2001-07-21' );
```

Update a Row

- Sometimes information changes.
- Sure, you could delete and insert another row, but it wouldn't have the same ID, and this could cause problems with the relationship of the information in this table to other tables.
- That's why you should write an `UPDATE` statement to correct the issue.
- For the first example, we're going to update a single row.
- As seen in the second and third examples, you can update multiple rows by using columns that have repeated values.

```
-- update the owner of pet 3 to 'Dan'
UPDATE pet
    SET owner = 'Dan'
    WHERE id = 3;

-- make all cats owned by 'Dan'

UPDATE pet
    SET owner = 'Dan'
    WHERE species = 'Cat';

-- make all dogs owned by Dave
UPDATE pet
    SET owner = 'Dave'
    WHERE species = 'Dog';
```

Delete a Row

- When records are no longer needed, the information should be removed from the database.
- The more records we have, the longer it takes queries to run and return data.
- For the first example, we're going to `delete` a single row.
- As seen in the second and third examples, you can delete multiple rows by using columns that have repeated values.

```
-- Remove the first pet from the table
DELETE FROM pet
    WHERE id = 1;

-- Remove all cats from the table
DELETE FROM pet
    WHERE species = 'CAT';

-- Remove all pets from the table Ben
owned
DELETE FROM pet
    WHERE owner = 'Ben';
```

Search for Rows

- So you have all of this data… now how are you going to be able to retrieve it?
- You'll need to `SELECT` it from the table!
- To get all columns, use the * command.
- However, in most queries, you will only pull the columns that you need.
- The more columns you include, the slower the query will run!

```
-- return all of the rows in the pet table
SELECT * FROM pet;

-- return all of Dan's pets
SELECT * FROM pet
    WHERE owner = 'Dan';

-- Return the pet name, sex, and birthday
-- for the pets that Dave owns
SELECT name, sex, birthday FROM pet
    WHERE owner = 'Dave';
```

Special Column Functions

- Maybe you want to find out how many pets someone has.
- Or, maybe you want to know what kinds of pets a person owns.
- SQL, and many other query languages have special functions that allow you to do more than just return the data.
- **DISTINCT** - Return only unique row values. Other languages use UNIQUE.
- **COUNT** - Return the number of rows that match the specified query

```
-- return the number of the rows in the pet table
SELECT COUNT(*) FROM pet;

-- return the number of Dan's pets
SELECT COUNT(*) FROM pet
    WHERE owner = 'Dan';

-- return the unique pet species from the table
SELECT DISTINCT(species) FROM pet;

-- return the unique number pet species
SELECT COUNT(DISTINCT(species)) FROM pet;
```

Relational Database Queries

- You can merge information from 2 tables to produce a more complete result set.
- Example : if you had an additional pets database table that contained personal information about the pet's owners, you could combine portions of both tables into a specific report
  - Perhaps a report on owners (name, address, contact information, etc. - all from a 2nd “owners” table along with information about their pets (perhaps names and species)

Relational Database Queries

- Let's assume we have a new table named `owners` that has a unique ID and that our `pets` table's owner column is also an ID related to the one in the owner's table.
- Then we could do something like:

```
SELECT owners.name, address, phoneNumber, pet.name,
    species FROM owners, pet
    WHERE owner.ID = pet.owner
```

- This allowed us to mesh the information from two tables together when you query to get a more complete results set.

Relational Database Queries

- Often we'd like our queries to come back sorted
- SQL (and MySQL) can do this easily for us using the `ORDER BY` command:

```
SELECT owners.name, address, phoneNumber, pet.name,
    species FROM owners, pet
    WHERE owner.ID = pet.owner
    ORDER BY owner.name ASC, pet.name ASC
```

Logical Operators

- SQL, like many languages, have logical operators that can be useful to get to the query that you desire.
- They're pretty similar to other languages that you've used in the past.
- Before we move on, it might be useful to review them.
- Example : Display the names of all male dogs (and their owners) that were born after 2005.

```
SELECT species, owner, birth
FROM pets WHERE birth >
12/31/2005 && sex = 'M'
```

```
-- logical and operators
AND, &&

-- logical not operators (negators)
NOT, !

-- logical or operators
||, OR

-- logical xor
XOR

-- you can also check whether a value is null or not
IS NULL, IS NOT NULL -- cannot use = or !=

-- you can also compare values

-- greater than, greater than or equal to
>, >=

-- less than, less than or equal to
<, <=
```

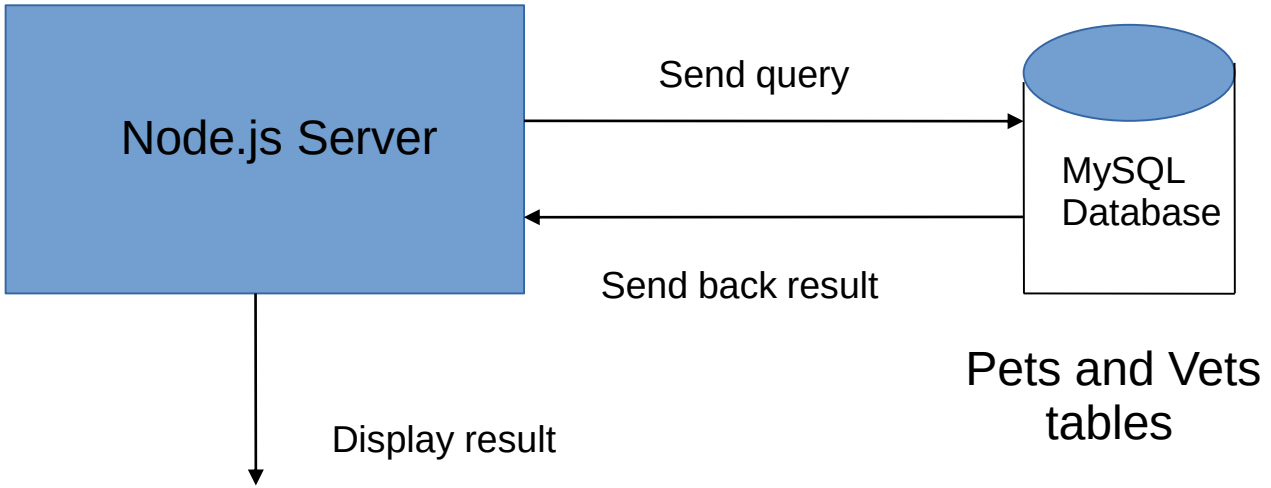
MySQL / Node.js Interactions

- We are now ready to discuss and demonstrate how a Noje.js script can be constructed to interact with a MySQL database to:
  - Send a query to the database
  - Parse and display the MySQL response
- Note that this is one of the final steps necessary before we can create fully integrated client-server applications
- The following slide illustrates a flow diagram of this process

Diagram of Information Flow
Node.js / MySQL Interaction

Flow Diagram (all pets owned by Dave)

SELECT owner, species, name from pet where owner = 'Dave'



Owner	Species	Pet name
Dave	Dog	Bella
Dave	Dog	Mickey
Dave	Cat	Lombard

Node.js Code

- To connect to a MySQL database you must install the `mysql` module
  - `npm install mysql`
- Then create an instance of this, populate connection information and try to connect!

```
var app = express();
var mysql = require('mysql');
var con = mysql.createConnection({
   host:  'localhost',
   user:  '<MYSQL username>',
   password: '<MYSQL_Password>',
   database: '<your_db_name>'
});
con.connect(function(err)  {
    if (err)  {
       console.log("Error connecting to database");
    }
    else  {
       console.log("Database successfully connected");
    }
 });
```

Node.js Code

- Now we can create and process queries.

```
 app.get("/", function(req, res)  {
    con.query('SELECT owner, species, name from pet where owner =
        \'Dave\'',
        function(err,rows,fields) {
            if (err)
                console.log('Error during query processing');
            else
                console.log('Here is the result : ', rows);
            // See next slide for an alternate way to parse
            //and display the result
        });
});
app.listen(3000);
```

Node.js Code

- We probably actually want to go through the rows of the results and do something with them…

```
for  (var i = 0; i < rows.length; i++) {
   console.log(rows[i].owner, rows[i].species,rows[i].name);
};
```