
THE UNIVERSITY
of ADELAIDE
Web and Database Computing
adelaide.edu.au
Server APIs and Authentication: Adding MySQL

Integrating Databases in Web Applications

Databases in Web Applications

Recall from Week 5 that Databases are applications that are optimized for storing and accessing data efficiently.

By now you should be familiar with SQL based relational databases and how we can interact with them through queries, but how do we connect them with our web applications?

Databases in Web Applications

The Node.js `mysql` module provides us with methods for interacting with our database.

To use it there's a few extra steps we'll need:

1. Install the module
2. Configure it for our RDBMS
3. Make it accessible to our rotues

Once it's setup, then we can query the database!

Installing the module

First we need to install the mysql module for our server

```
npm install --save mysql
```

Setup connection to RDBMS in Express

Need to setup the mysql in `app.js` (not included by default), and create middleware for it

```
...

// use mysql in this app
var mysql = require('mysql');

// create a 'pool' (group) of connections to be used for connecting with our SQL server
var dbConnectionPool = mysql.createPool({
  host: 'localhost',
  database: 'blog'
});

var app = express();

// Middleware for accessing database: We need access to the database to be available *before* we process routes in index.js,
// so this code needs to be *before* the app.use('/', routes);
// Express will run this function on every request and then continue with the next module, index.js.
// So for all requests that we handle in index.js, we'll be able to access the pool of connections using req.pool
app.use(function(req, res, next) {
  req.pool = dbConnectionPool;
  next();
});

app.use(logger('dev'));
app.use(express.json());
...
```

Accessing the database connection

Now that we've configured our express server to be able to connect to the database, we can use that configuration in our routes.

```
router.get('/result', function(req, res) {
  //Connect to the database
  req.pool.getConnection(function(err,connection) {
    if (err) {
      res.sendStatus(500);
      return;
    }
  });
});
```

- Recall that our middleware sets req.pool to our database pool that we created in the app.js file.
- `getConnection()` is a function in the mysql module that connects to the mysql database.
- The callback function will have `err` and `connection` as parameters.
  - `connection` is the connection to the database.

Querying the database

Let's run a basic query

```
router.get('/result', function(req, res, next) {
  //Connect to the database
  req.pool.getConnection( function(err,connection) {
    if (err) {
      res.sendStatus(500);
      return;
    }
    var query = "SHOW TABLES";
    connection.query(query, function(err, rows, fields) {
      connection.release(); // release connection
      if (err) {
        res.sendStatus(500);
        return;
      }
      res.json(rows); //send response
    });
  });
});
```

Querying the database: What's happening?

```
var query = "SHOW TABLES";
connection.query(query, function(err, rows, fields) {
  connection.release(); // release connection
  if (err) {
    res.sendStatus(500);
    return;
  }
  res.json(rows); //send response
});
```

- Once we've established a connection, we can run one or more queries using `connection.query()`.
- The `connection.query()` method takes
  - a SQL query string (the same as those you've been writing),
  - and the function to run when the query returns results.
- Once the query is completed, the callback function is run.
  - If we are done with database for this request, we should close it immediately; this will allow other requests to access the database.
- The callback function will have `err`, `rows` and `fields` as parameters.
  - `rows` is an array of objects containing the results of the query.

Querying the database: Prepared Statements

What if we want to use information from the request in our query?

```
var username = req.query.username; // if username was a field in the request URL
var query = "SELECT date,contents FROM posts WHERE author = ?";
connection.query(query, [username], function(err, rows, fields) {
  connection.release(); // release connection
  if (err) {
    res.sendStatus(500);
    return;
  }
  res.json(rows); //send response
});
```

- The `connection.query()` method takes
  - a SQL query string,
  - **an array of parameters** that we want to include as part of the query,
  - and the function to run when the query returns results.
- If the query requires data from the client to run (or other external sources), `?` placeholders are used.
  - The `?` placeholders are replaced with the values in the parameters array.
  - Using placeholders like this keeps our SQL secure. These are called **prepared statements**.

Query response format

In the function on the previous slide, we've sent the rows selected by the query directly on to the client:

```
connection.query(query, function(err, rows, fields) {
  connection.release(); // release connection
  if (err) {
    res.sendStatus(500);
    return;
  }
  res.json(rows); //send response
});
```

What does this data look like and how can we use it on our client?

An array of RowDataPacket objects that we can send as JSON:

```
[ RowDataPacket { date: 2016-01-23T10:45:48.000Z,
                  contents: 'Lorem ipsum dolor' },
  RowDataPacket { date: 2016-02-15T02:51:32.000Z,
                  contents: 'Phasellus aliquam, leo' } ]
```

Query response format

```
[ RowDataPacket { date: 2016-01-23T10:45:48.000Z,
                  contents: 'Lorem ipsum dolor' },
  RowDataPacket { date: 2016-02-15T02:51:32.000Z,
                  contents: 'Phasellus aliquam, leo' } ]
```

How do we use this data in code?

Same way we access other values in objects, for example:

- rows[0].contents will give us the string `'Lorem ipsum dolor'`
- rows[1].date will give us the date `2016-02-15T02:51:32.000Z`

Or, as seen before, we can send the response straight to the client as JSON:

- `res.json(rows);`
- The `RowDataPacket` information is stripped away.
- Once converted back from JSON, the client can access the data the same way as described above.

Summary

- NodeJS can interact with MySQL/MariaDB and other databases using the appropriate module.
- For a SQL server we need to
  - Install the module
  - Setup the connection and middleware
- Once setup we can run queries
  - Use prepared statements to secure your queries.
  - The response is a JavaScript object that be be used or converted to JSON and sent.

THE UNIVERSITY
of ADELAIDE