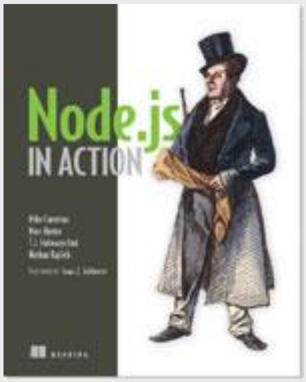


Node.js in Action: Relational database management systems

By Mike Cantelon

Developers have many relational database options, but most choose open source databases, primarily because they're well supported, they work well, and they don't cost anything. In this article, we'll look at MySQL, one of the two most popular full-featured relational databases.

Relational database management systems (RDBMSs) allow complex information to be stored and easily queried. RDBMSs have traditionally been used for relatively high-end applications, such as content management, customer relationship management, and shopping carts. They can perform well when used correctly, but they require specialized administration knowledge and access to a database server. They also require knowledge of SQL, although there are object-relational mappers (ORMs) with APIs that can write SQL for you in the background. RDBMS administration, ORMs, and SQL are beyond the scope of this article, but you'll find many online resources that cover these technologies.

Developers have many relational database options, but most choose open source databases, primarily because they're well supported, they work well, and they don't cost anything. In this article, we'll look at MySQL, one of the two most popular full-featured relational databases. MySQL and PostgreSQL have similar capabilities, and both are solid choices. If you haven't used either, MySQL is easier to set up and has a larger user base.

Let's look at MySQL.

MySQL

MySQL is the world's most popular SQL database, and it's well supported by the Node community. If you're new to MySQL and interested in learning about it, you'll find the official tutorial online (http://dev.mysql.com/doc/refman/5.0/en/tutorial.html). For those new to SQL, many online tutorials and books, including Chris Fehily's *SQL: Visual QuickStart Guide* (Peachpit Press, 2008), are available to help you get up to speed.

USING MYSQL TO BUILD A WORK-TRACKING APP

To see how Node takes advantage of MySQL, let's look at an application that requires an RDBMS. Let's say you're creating a serverless web application to keep track of how you spend your workdays. You'll need to record the date of the work, the time spent on the work, and a description of the work performed.

The application you'll build will have a form in which details about the work performed can be entered, as shown in figure 1.

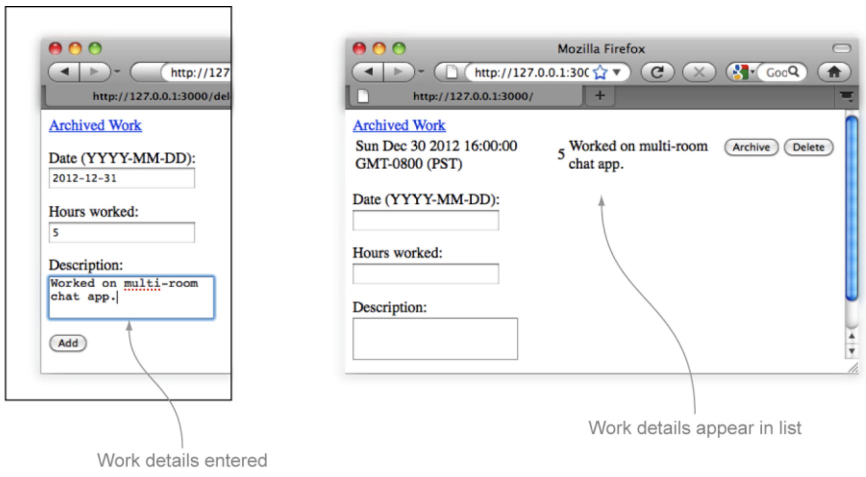


Figure 1 Recording details of work performed

Once the work information has been entered, it can be archived or deleted so it doesn't show above the fields used to enter more work, as shown in figure 2. Clicking the Archived Work link will then display any work items that have been archived.

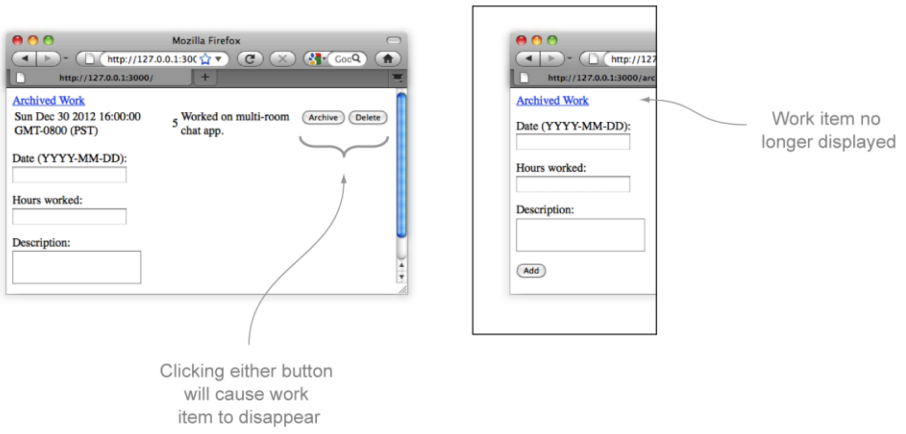

Mozilla Firefox
http://127.0.0.1:3000/
Archived Work
Sun Dec 30 2012 16:00:00 GMT-0800 (PST)
5
Worked on multi-room chat app.
Archive
Delete
Date (YYYY-MM-DD):
Hours worked:
Description:
Add
Work item no longer displayed
Clicking either button will cause work item to disappear

Figure 2 Archiving or deleting details of work performed

You could build this web application using the filesystem as a simple data store, but it would be tricky to build reports with the data. If you wanted to create a report on the work you did last week, for example, you'd have to read every work record stored and check the record's date. Having application data in an RDBMS gives you the ability to generate reports easily using SQL queries.

To build a work-tracking application, you'll need to do the following:

- Create the application logic
- Create helper functions needed to make the application work
- Write functions that let you add, delete, update, and retrieve data with MySQL
- Write code that renders the HTML records and forms

The application will leverage Node's built-in http module for web server functionality and will use a third-party module to interact with a MySQL server. A custom module named *timetrack* will contain application-specific functions for storing, modifying, and retrieving data using MySQL. Figure 3 provides an overview of the application.

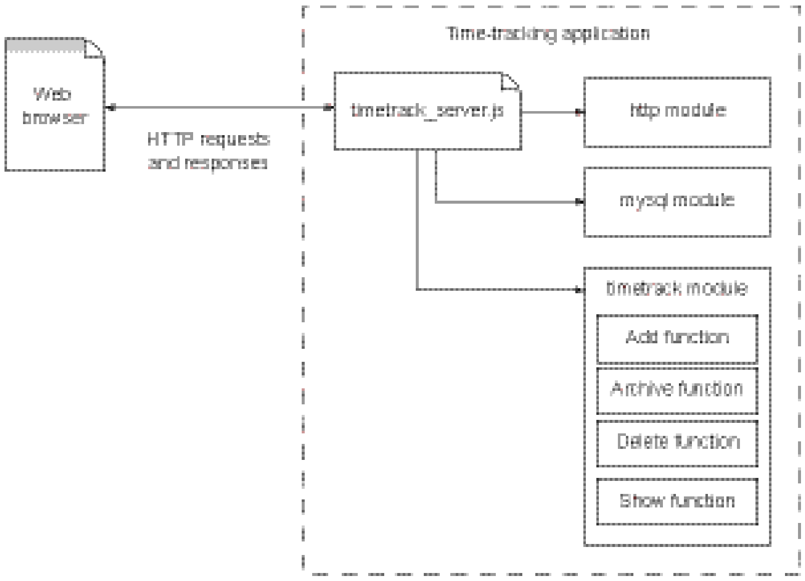
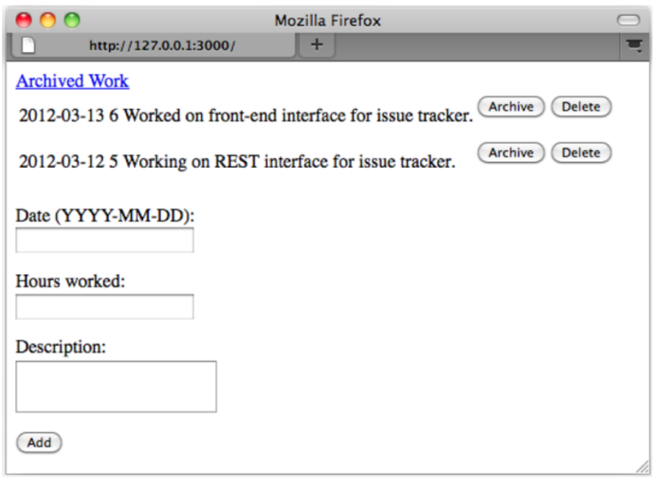


Figure 3 How the work-tracking application will be structured

The result, as shown in figure 4, will be a simple web application that allows you to record work performed and review, archive, and delete the work records.



Archived Work

2012-03-13 6 Worked on front-end interface for issue tracker.

2012-03-12 5 Working on REST interface for issue tracker.

Date (YYYY-MM-DD):

Hours worked:

Description:

Figure 4 A simple web application that allows you to track work performed

To allow Node to talk to MySQL, we'll use Felix Geisendörfer's popular node-mysql module (https://github.com/felixge/node-mysql). To begin, install the MySQL Node module using the following command:

```
npm install mysql@2.5.4
```

CREATING THE APPLICATION LOGIC

Next, you need to create two files for application logic. The application will be composed of two files: timetrack_server.js, used to start the application, and timetrack.js, a module containing application-related functionality.

To start, create a file named timetrack_server.js and include the code in listing 1. This code includes Node's HTTP API, application-specific logic, and a MySQL API. Fill in the `host`, `user`, and `password` settings with those that correspond to your MySQL configuration.

Listing 1 Application setup and database connection initialization

```
var http = require('http');
var work = require('./lib/timetrack');
var mysql = require('mysql');                            #A
var db = mysql.createConnection({                        #B
  host:     '127.0.0.1',
  user:     'myuser',
  password: 'mypassword',
  database: 'timetrack'
});
```

#A Require MySQL API
#B Connect to MySQL

Next, add the logic in listing 2 to define the basic web application behavior. The application allows you to browse, add, and delete work performance records. In addition, the app will let you archive work records. Archiving a work record hides it on the main page, but archived records remain browsable on a separate web page.

Listing 2 HTTP request routing

```
var server = http.createServer(function(req, res) {
  switch (req.method) {
    case 'POST':                                                #A
      switch(req.url) {
        case '/':
          work.add(db, req, res);
          break;
        case '/archive':
          work.archive(db, req, res);
          break;
        case '/delete':
          work.delete(db, req, res);
          break;
      }
      break;
    case 'GET':                                              #B
      switch(req.url) {
        case '/':
          work.show(db, res);
          break;
        case '/archived':
          work.showArchived(db, res);
      }
      break;
  }
});
```

#A Route HTTP POST requests
#B Route HTTP GET requests

The code in listing 3 is the final addition to timetrack_server.js. This logic creates a database table if none exists and starts the HTTP server listening to IP address 127.0.0.1 on TCP/IP port 3000. All node-mysql queries are performed using the `query` function.

Listing 3 Database table creation

```
db.query(
  "CREATE TABLE IF NOT EXISTS work ("                       #A
  + "id INT(10) NOT NULL AUTO_INCREMENT, "
  + "hours DECIMAL(5,2) DEFAULT 0, "
  + "date DATE, "
  + "archived INT(1) DEFAULT 0, "
  + "description LONGTEXT,"
  + "PRIMARY KEY(id))",
  function(err) {
    if (err) throw err;
    console.log('Server started...');
    server.listen(3000);                      #B
  }
);
```

#A Table-creation SQL
#B Start HTTP server

CREATING HELPER FUNCTIONS THAT SEND HTML, CREATE FORMS, AND RECEIVE FORM DATA

Now that you've fully defined the file you'll use to start the application, it's time to create the file that defines the rest of the application's functionality. Create a directory named lib, and inside this directory create a file named timetrack.js. Inside this file, insert the logic from listing 4, which includes the Node querystring API and defines helper functions for sending web page HTML and receiving data submitted through forms.

Listing 4 Helper functions: sending HTML, creating forms, receiving form data

```
var qs = require('querystring');
exports.sendHtml = function(res, html) {                             #A
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('Content-Length', Buffer.byteLength(html));
  res.end(html);
};
exports.parseReceivedData = function(req, cb) {                         #B
  var body = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk){ body += chunk });
  req.on('end', function() {
    var data = qs.parse(body);
    cb(data);
  });
};
exports.actionForm = function(id, path, label) {                   #V
  var html = '<form method="POST" action="' + path + '">' +
    '<input type="hidden" name="id" value="' + id + '">' +
    '<input type="submit" value="' + label + '" />' +
    '</form>';
  return html;
};
```

#A Send HTML response
#B Parse HTTP POST data
#C Render simple form data

ADDING DATA WITH MYSQL

With the helper functions in place, it’s time to define the logic that will add a work record to the MySQL database. Add the code in the next listing to timetrack.js.

Listing 5 Adding a work record

```
exports.add = function(db, req, res) {
  exports.parseReceivedData(req, function(work) {                  #A
    db.query(
      "INSERT INTO work (hours, date, description) " +             #B
      " VALUES (?, ?, ?)",
      [work.hours, work.date, work.description],                 #C
      function(err) {
        if (err) throw err;
        exports.show(db, res);                                 #D
      }
    );
  });
};
```

#A Parse HTTP POST data
#B SQL to add work record
#C Work record data
#D Show user a list of work records

Note that you use the question mark character (`?`) as a placeholder to indicate where a parameter should be placed. Each parameter is automatically escaped by the `query` method before being added to the query, preventing SQL injection attacks.

Note also that the second argument of the `query` method is now a list of values to substitute for the placeholders.

DELETING MYSQL DATA

Next, you need to add the following code to timetrack.js. This logic will delete a work record.

Listing 6 Deleting a work record

```
exports.delete = function(db, req, res) {
  exports.parseReceivedData(req, function(work) {                  #A
    db.query(
      "DELETE FROM work WHERE id=?",                            #B
      [work.id],                                                #C
      function(err) {
        if (err) throw err;
        exports.show(db, res);                                     #D
      }
    );
  });
};
```

#A Parse HTTP POST data
#B SQL to delete work record
#C Work record ID
#D Show user a list of work records

UPDATING MYSQL DATA

To add logic that will update a work record, flagging it as archived, add the following code to timetrack.js.

Listing 7 Archiving a work record

```
exports.archive = function(db, req, res) {
  exports.parseReceivedData(req, function(work) {          #A
    db.query(
      "UPDATE work SET archived=1 WHERE id=?",                #B
      [work.id],                                            #C
      function(err) {
        if (err) throw err;
        exports.show(db, res);                              #D
      }
    );
  });
};
```

#A Parse HTTP POST data
#B SQL to update work record
#C Work record ID

#D Show user a list of work records

RETRIEVING MYSQL DATA

Now that you've defined the logic that will add, delete, and update a work record, you can add the logic in listing 8 to retrieve work-record data—archived or unarchived—so it can be rendered as HTML. When issuing the query, a callback is provided that includes a `rows` argument for the returned records.

Listing 8 Retrieving work records

```
exports.show = function(db, res, showArchived) {
  var query = "SELECT * FROM work " +                          #A
    "WHERE archived=? " +
    "ORDER BY date DESC";
  var archiveValue = (showArchived) ? 1 : 0;
  db.query(
    query,
    [archiveValue],                                  #B
    function(err, rows) {
      if (err) throw err;
      html = (showArchived)
        ? ''
        : '<a href="/archived">Archived Work</a><br/>';
      html += exports.workHitlistHtml(rows);                   #C
      html += exports.workFormHtml();
      exports.sendHtml(res, html);                             #D
    }
  );
};
exports.showArchived = function(db, res) {
  exports.show(db, res, true);                               #E
};
```

#A SQL to fetch work records
#B Desired work-record archive status
#C Format results as HTML table
#D Send HTML response to user
#E Show only archived work records

RENDERING MYSQL RECORDS

Add the logic in the following listing to timetrack.js. It'll do the rendering of work records to HTML.

Listing 9 Rendering work records to an HTML table

```
exports.workHitlistHtml = function(rows) {
  var html = '<table>';
  for(var i in rows) {                                         #A
    html += '<tr>';
    html += '<td>' + rows[i].date + '</td>';
    html += '<td>' + rows[i].hours + '</td>';
```

```
      html += '<td>' + rows[i].description + '</td>';
      if (!rows[i].archived) {                                    #B
        html += '<td>' + exports.workArchiveForm(rows[i].id) + '</td>';
      }
      html += '<td>' + exports.workDeleteForm(rows[i].id) + '</td>';
      html += '</tr>';
    }
    html += '</table>';
    return html;
  };
```

#A Render each work record as HTML table row
#B Show archive button if work record isn't already archived

RENDERING HTML FORMS

Finally, add the following code to timetrack.js to render the HTML forms needed by the application.

Listing 10 HTML forms for adding, archiving, and deleting work records

```
exports.workFormHtml = function() {
  var html = '<form method="POST" action="/">' +                       #A
    '<p>Date (YYYY-MM-DD):<br/><input name="date" type="text"><p/>' +
    '<p>Hours worked:<br/><input name="hours" type="text"><p/>' +
    '<p>Description:<br/>' +
    '<textarea name="description"></textarea></p>' +
    '<input type="submit" value="Add" />' +
    '</form>';
  return html;
};
exports.workArchiveForm = function(id) {                                #B
  return exports.actionForm(id, '/archive', 'Archive');
};
exports.workDeleteForm = function(id) {                                #C
  return exports.actionForm(id, '/delete', 'Delete');
};
```

#A Render blank HTML form for entering new work record
#B Render Archive button form
#C Render Delete button form

TRYING IT OUT

Now that you've fully defined the application, you can run it. Make sure that you've created a database named timetrack using your MySQL administration interface of choice. Then start the application by entering the following into your command line:

```
node timetrack_server.js
```

Finally, navigate to http://127.0.0.1:3000/ in a web browser to use the application.