Secure Your Node.js Web Application

Keep Attackers Out and Users Happy

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Secure Your Node.js Web Application

Keep Attackers Out and Users Happy

Karl Düüna

edited by Fahmida Y. Rashid

Secure Your Node.js Web Application

Keep Attackers Out and Users Happy

Karl Düüna

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Fahmida Y. Rashid (editor)
Potomac Indexing, LLC (index)
Linda Recktenwald (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-085-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2016

Real knowledge is to know the extent of one's ignorance.

➤ *Confucius*

CHAPTER 5

Secure Your Database Interactions

In the last chapter we covered how to identify code injection attacks and ways to defend your server and processes. We'll continue by learning about database security and especially about how most injection attacks target it.

The database is the heart of most modern web applications—without it the applications are just empty husks. Books with covers but no pages. Application data stored in the database, especially user information, is a prime target for attackers. They want the passwords and personal information to log into accounts on other sites. They desire financial and credit card information to empty user bank accounts. They may want sensitive—or potentially embarrassing—information that can be used to blackmail users.

Knowledge is power.

Many companies suffered data breaches in 2014, and millions of individuals saw their Social Security numbers, credit card numbers, and passwords stolen. While many of these attacks, especially high-profile ones, used highly sophisticated methods, some used a far simpler *database injection attack*. This is similar to shell injection, except the targeted execution layer is the database. As mentioned in the previous chapter, OWASP (Open Web Application Security Project)[1] considers database injection the top attack vector against web applications.

We want to stop attackers from getting their hands on our users' data. If the attackers succeed, we lose our users' trust, our business won't be successful, and we won't be able to afford that new Tesla we like so much. We can't have that, so let's spend this chapter preventing database injection and related attack vectors.

1. https://www.owasp.org

You can work work through the examples in this chapter with MySQL,[2] PostgreSQL,[3] and MongoDB[4] databases. We'll walk through the data model and table schemas together. Setting up the database and account is out of scope for this book, so you should reference your database documentation to get started.

Start with the Basics: Set Up the Database

Let's start from the beginning—your application has to successfully connect to the database before you can do anything. Let's make sure you don't fumble the ball even before you cross the line of scrimmage.

Any database that you work with, and it doesn't matter if we're talking about MySQL, Mongo, Redis, or any other database system, should be configured to use authenticated users. Sometimes people don't bother with user accounts and let everyone (including applications) connect to the database without a password. They typically block outside connections, which is a good thing to do. Unfortunately, it's not sufficient, even if the database lives on the same machine.

Yes, blocking outside connections narrows the attack surface significantly. But the attacker can bypass this restriction by gaining access to one of the whitelisted machines or the IP addresses. If I'm running a Redis database server on my machine with default settings enabled—no authentication—then all the attacker has to do is somehow get onto my machine. It doesn't matter if the attacker is using an unprivileged account since there's no barrier to connecting to the database. Voila! Full access to every database and all the data.

Imagine your server as an apartment building, individual apartments as databases, and each room in the apartment as a table in the database. Just because you have a lock on the front door of the building does not mean you don't want locks for each apartment. It would still disturb you very much if someone from the street managed sneak through the front door and then could roam around the building visiting every room and looking at your things. Just because someone has access to the server doesn't mean they should have access to all the data in each database.

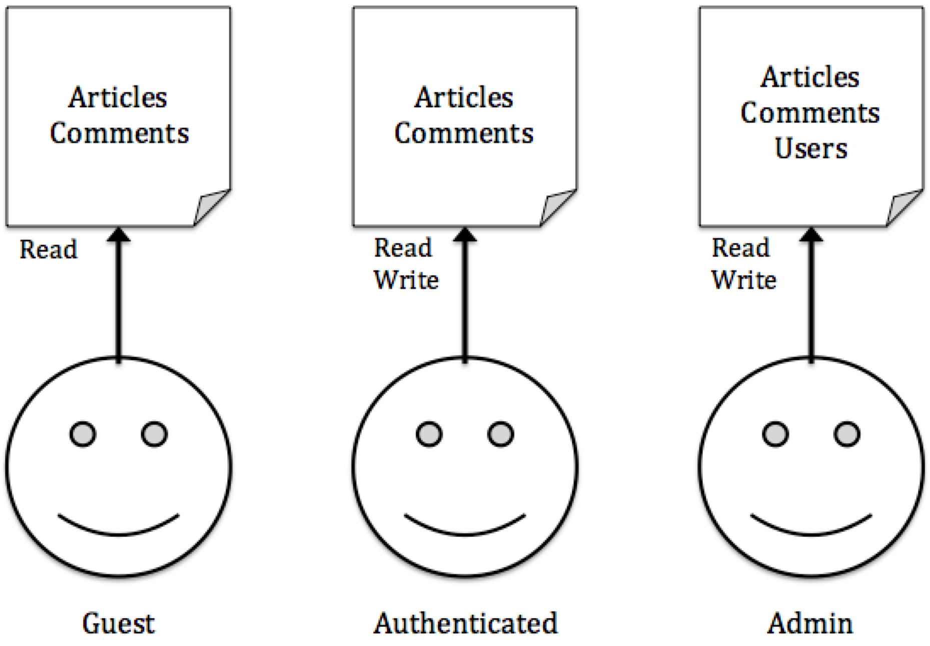
It's clear that proper authentication is an important aspect of defense. It will be harder for attackers to pull off a successful attack if the application is

2. http://www.mysql.com/
3. http://www.postgresql.org/
4. http://www.mongodb.org/

smart about who it lets in, who it keeps out, and who is allowed to do what. Setting up authentication also lets you follow the principle of least privilege to fine-tune different levels of access for each account. You may want to revisit *Follow the Principle of Least Privilege*, on page ? for a refresher.

Our theoretical web application has three separate database accounts for users—guest, authenticated, and admin; see the following graphic.

- Guest users can only read articles on the site—so the guest database account needs only read access for the database tables.
- Authenticated users can read and write articles, as well as post comments—meaning the authenticated database account should have read and write access on tables related to articles and comments.
- Admin users can add new users and do other administrative tasks. The admin account on the database has the highest privilege level, with read/write privileges on most, if not all, tables.



You might wonder why you need to go through the trouble of having separate database accounts if you already have separate user roles. Suppose you have a database injection vulnerability somewhere in the guest section of the application. Attackers who exploit this hole won't be able to cause as much damage because the guest database user has only read privileges on the tables.

Your admin account may not even need all the privileges it currently has. You probably would never drop tables from the web application, for example. Remove that privilege from the admin database account and the attacker won't be able to use code injection to delete data. You'd still have elevated privileges as the admin user when connected directly to the database, which is all you need.

Defining multiple accounts in the database with various levels of privileges is a good thing, but you need to use them. Look at the following code snippet to see an example how you could manage multiple connections to the database:

chp-5-database/mysql-multiple-connections.js

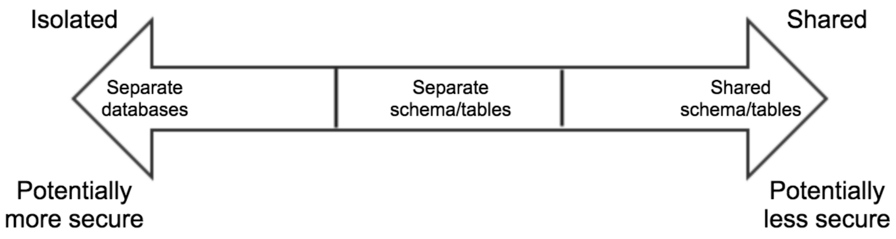
```
// Set up guest connection
var guestConnection = mysql.createConnection({
    host     : 'localhost',
    user     : args.gu,
    database : args.d,
    password : args.gp,

    // Set for testing, do not do unless you have a good reason
    multipleStatements: true
});
guestConnection.connect();

// Set up admin connection
var adminConnection = mysql.createConnection({
    host     : 'localhost',
    user     : args.au,
    database : args.d,
    password : args.ap,

    // Set for testing, do not do unless you have a good reason
    multipleStatements: true
});
adminConnection.connect();

// Middleware for checking the logged in status
app.use(function (req, res, next) {
    // If we have an admin session then attach adminConnection
    if(req.session && req.session.isAdmin) {
        req.db = adminConnection;
    }
    // Otherwise attach guestConnection
    else {
        req.db = guestConnection;
    }
    next();
});
```

You now have a lock on your database, and, more importantly, you should now understand why you need one. The lock will deter people trying to snoop through data they shouldn't be allowed to see, which will keep the clients happy. And that keeps *you* happy.

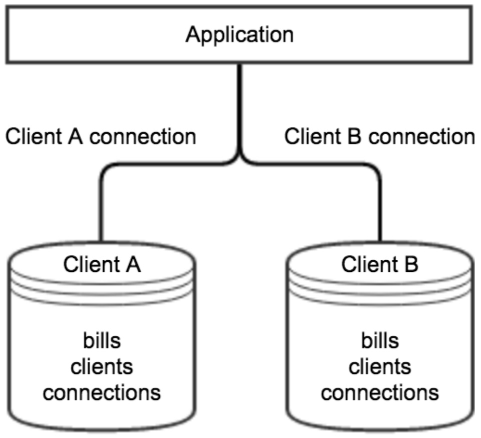
Let's take a quick look at another important data-separation paradigm that you should know besides simple role-based connection—multi-tenancy.

Separate Databases for Better Security

Many web applications serve as platforms for multiple clients at the same time. Depending on the application, this will involve storing business data or logic in the database. For example, you may have a CRM system where clients store their own records, connections, and billing information. You need to safely separate the data so that clients can't access each other's data. There are many approaches ranging from totally isolated databases to fully shared ones, but they tend to fall somewhere along the spectrum, as shown here.

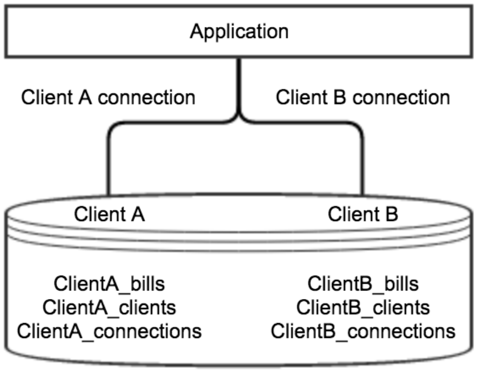


The security of these methods can vary and depend on other development and infrastructure requirements. But let's look at each one in detail.



First up is having separate databases for everyone, as shown in the graphic. This is the most isolated approach. Every client has a separate database for its data and each client can customize the data structure for its own needs. It can be more secure, but it comes with higher infrastructure costs. Databases take up space (an empty MongoDB database takes 32 MB, for example), and there's only so much space available on a server.

Completely isolating your tenants is a good approach when you have strict security needs and clients who are willing to pay extra for security. You can use this approach with most SQL and NoSQL databases (including MongoDB).



The middle approach is to use the same database but separate the schemas (tables/collections) for each client, as the graphic shows. Each tenant has access to only its own set of tables. Like the isolated approach, each tenant can customize the data structures and keep database connection levels separate from everyone else.

This method makes backups tricky. In the isolated approach, you back up and restore a single tenant's database without affecting anyone else. Here, you have to back up all the tables together, regardless of client. Restoring a single tenant's data is a challenge as you don't want to affect others.

Neither can you fully use this approach in MongoDB because it doesn't have collection-level access control. While there are modules like mongoose-multitenant,[5] you still can't control connection access at the collection level. You'd miss out on most of the security benefits of this approach, and MongoDB has limits on how many collections you can create in a single database as well.

The last approach is the most common—storing all the clients' data in the same database and sharing the database schema. The data is separated by providing a unique tenant identifier for each row. It's the cheapest approach because it has the lowest infrastructure requirements, but it has the highest implementation cost for security. You have to handle security in your code and manage all the data-separation mechanisms yourself.

Most applications start out from a shared model, since businesses generally start thinking about multi-tenancy only after the application has gotten large enough. Or they have to comply with regulatory requirements. There's no best approach, because that depends on your specific needs. I recommend generally starting out with the shared approach and eventually graduating to the isolated approach when you can.

Next, let's look at how to mitigate attacks against data stored in your database (or databases).

5. https://www.npmjs.com/package/mongoose-multitenant

Identify Database Injection Points in Your Code

We briefly talked about database injection earlier. It's a variation of code injection, but the intended target is the back-end database and not the application server. Let's look at this widely used attack in detail and discuss ways to prevent it.

If an application has code injection issues, it means the application is not correctly validating all input fields on the site. The same thing applies to database injection. Attackers enter a series of database commands into the application's input fields (such as a textbox in a blog's comment form) to trick the application into executing the commands within the database. If the application builds its database queries by concatenating user input with hardcoded strings instead of using a decent ORM (object-relational mapper) and neglects to properly escape input data, then the attacker succeeds.

Let's take a look at what a database injection flaw looks like. Make sure you understand what's happening here, because we'll revisit this again in later chapters.

For the following examples you'll need a MySQL database. For all database examples, we'll use the minimist[6] module to parse command-line arguments—which we'll use to supply the database connection information.

chp-5-database/mysql-exploitable.js

```
var mysql = require('mysql');
var express = require('express');
var args = require('minimist')(process.argv);

if(!args.u || !args.d || !args.p) {
    console.log('This example requires the ' +
    '-u (user), ' +
    '-d (mysql db) and ' +
    '-p (password) command line variables');

    process.exit();
}

var connection = mysql.createConnection({
    host     : 'localhost',
    user     : args.u,
    database : args.d,
    password : args.p,
    multipleStatements: true // This is so we can execute multiple statements
});
```

6. https://github.com/substack/minimist

```
connection.connect();

var app = express();

app.get('/', function (req, res) {
    res.send('ok');
});

app.get('/:name', function(req, res, next){

    // Query the account based on url parameters
    // As you can see we use no validation on the name parameter
    connection.query('SELECT * FROM accounts WHERE name="' + req.params.name + '"',
        function(err, rows, fields) {
            if (err) {
                next(err);
                return;
            }
            res.send(JSON.stringify(rows));
        });
});

app.listen(3000);
```

Let's ignore the fact that this application doesn't perform any user validation. You now have an application that displays the account information you requested via the name parameter in JSON format. The database dump would look like the following:

```
CREATE TABLE `accounts` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `name` varchar(255) NOT NULL,
    `email` varchar(255) NOT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=3 ;

INSERT INTO `accounts` (`id`, `name`, `email`) VALUES
(1, 'karl', 'karl@dyyna.com'),
(2, 'juhan', 'juhan@gmail.com');
```

When you visit the URL /karl in the browser, you should see the following response:

```
[{
    "id": 1,
    "name": "karl",
    "email": "karl@dyyna.com"
}]
```

We could also construct the following URL to send the following SQL commands in a single line: /";SELECT%20*%20FROM%20accounts%20WHERE%20"1"="1. You can see here how the URL turns into two database queries:

```
SELECT * FROM accounts WHERE name="";SELECT * FROM accounts WHERE "1"="1";`
```

The first statement will return nothing because we didn't provide a value for name. However, the second statement will dump all the accounts stored in the database:

```
[
  [], // Our first query that selected nothing, since we terminated it
  [{
    "id": 1,
    "name": "karl",
    "email": "karl@dyyna.com"
  }, {
    "id": 2,
    "name": "juhan",
    "email": "juhan@gmail.com"
  }]
]
```

A select query is only the beginning of what we can do. Imagine the damage we could cause with /";DROP%20TABLE%20accounts;, which would delete the accounts table altogether. A database injection lets someone modify the commands sent to the database in order to view or modify the saved data. So how do you protect your application?

Good News About Running Multiple Queries in MySQL

The good news is that one of the most popular MySQL drivers for Node.js, node-mysql,[7] disables by default the ability to execute multiple commands in a single query. This makes SQL injection attacks much harder to launch on web applications if you use this module. The module won't let the attacker terminate the original query to start a separate malicious one. But for the sake of providing examples, we'll enable the multiple queries option in node-mysql.

Avoid SQL Injection Attacks

As you just saw, attackers can cause a lot of damage with database injection. We'll now look at four main defense methods to protect against this kind of attack: controlled error messages, input validation, escaping, and prepared statements.

7. https://github.com/felixge/node-mysql

Database injection attacks are divided into two types: blind and normal SQL injection. In normal SQL injection the attacker will see helpful error messages and/or the result of the attack on the web page. With blind SQL injection, the attacker sees only generic error messages if something is not valid.

In case of blind SQL injection attackers frequently append true-false statements such as `and 1=2` or `and 1=1` at the end of the query to see the different messages associated with successful and unsuccessful attempts. Blind SQL injection requires the attacker to collect the information needed one piece at a time.

The mechanics of defending are the same, but blind SQL injection is much harder and more time consuming for the attacker to pull off. This is why your first line of defense is to handle errors properly, as previously discussed in *Limit Error Messages in Production*, on page ?. Forcing the attacker to spend more time determining whether there is a vulnerability and how to get at the data benefits you.

The second step is validating user input. You'll have to verify that user-entered data falls within expected parameters and is not malicious. Say you're manually constructing MySQL commands by combining user-entered data with queries hardcoded within the application code. This approach is very important if you're using a database driver and handling database queries this way. You have to be attentive when checking every input to make sure users aren't entering malicious strings. The best approach is *whitelisting*, or allowing only types of data you expect to see.

Let's go back to the previous example and see how you can validate user input:

chp-5-database/mysql-exploitable-validation.js

```
var connection = mysql.createConnection({
    host     : 'localhost',
    user     : args.u,
    database : args.d,
    password : args.p,
    multipleStatements: true // This is so we can execute multiple statements
});
connection.connect();

var app = express();

app.get('/', function (req, res) {
    res.send('ok');
});
```

```
app.get('/:name', function(req, res, next){
➤   // Validate that the name has only letters
➤   if(req.params.name.match(/[^a-zA-Z]/)) {
➤       // It didn't so send a Bad Request response
➤       res.sendStatus(400);
➤       return;
➤   }

    // Query the account based on url parameters
    connection.query('SELECT * FROM accounts WHERE name="' + req.params.name + '"',
        function(err, rows, fields) {
            if (err) {
                next(err);
                return;
            }
            res.send(JSON.stringify(rows));
        });
});
```

This is restrictive but efficient. The thing is, you can't take this approach all the time because you don't always have such a clear understanding of what would be considered valid input.

Let's move on to the third approach, escaping. This means that all characters that can potentially break the query are formatted in such a way that the application doesn't treat them as part of a command.

This is a widely used method and many libraries, including node-mysql, provide ready-to-go functions for escaping well-known problem characters. You can utilize connection.escape(), which is the Node equivalent of PHP's mysqli_escape_string(). This way you don't have the hassle of trying to write the function yourself since you can just use a well-tested one.

So modify your vulnerable example again to escape the input string:

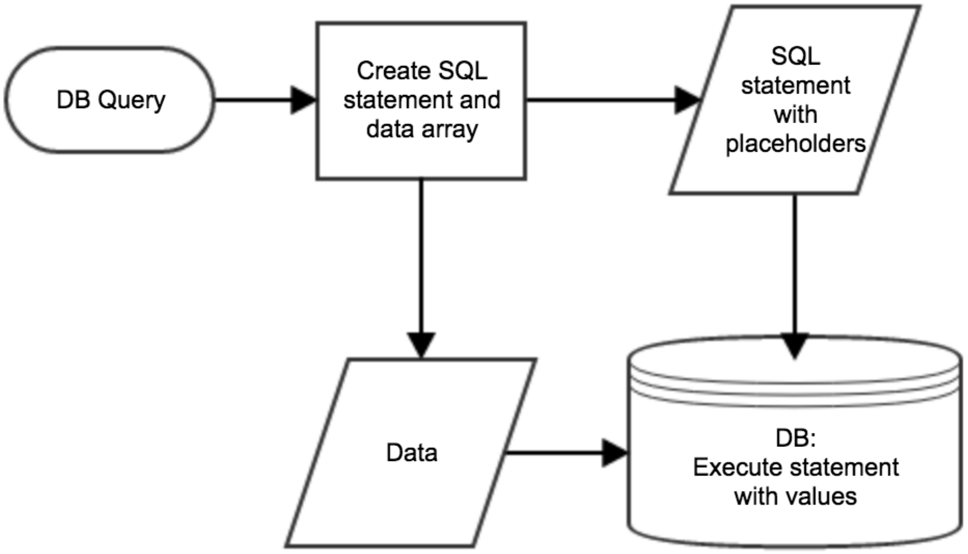
chp-5-database/mysql-exploitable-fixed.js

```
var connection = mysql.createConnection({
    host     : 'localhost',
    user     : args.u,
    database : args.d,
    password : args.p,
    multipleStatements: true // This is so we can execute multiple statements
});
connection.connect();

var app = express();

app.get('/', function (req, res) {
    res.send('ok');
});
```

```
app.get('/:name', function(req, res, next){
➤     // Query the account based on url parameters
➤     var query = 'SELECT * FROM accounts WHERE name="' +
➤         connection.escape(req.params.name) + '"';
➤
➤     connection.query(query, function(err, rows, fields) {
          if (err) {
              next(err);
              return;
          }
          res.send(JSON.stringify(rows));
      });
  });
```

The previous attack string no longer works because the quotation marks are escaped. The application now knows the quotes should be treated as part of a string and not as part of a command.

The final method is to use prepared statements; see the following illustration. Here, you completely separate the command and data parts of the query by sending them to the database separately. This leaves no room for misinterpretation and is a good way to protect against injection. As a bonus, it also provides a speed boost on queries that run many times because you can reuse the same procedure.


DB Query
Create SQL statement and data array
SQL statement with placeholders
Data
DB: Execute statement with values

Try out this technique:

chp-5-database/mysql-exploitable-fixed-prep.js

```
var connection = mysql.createConnection({
    host     : 'localhost',
    user     : args.u,
    database : args.d,
    password : args.p,
    multipleStatements: true // This is so we can execute multiple statements
});
connection.connect();

var app = express();

app.get('/', function (req, res) {
    res.send('ok');
});

app.get('/:name', function(req, res, next){

➤   // Query the account based on url parameters
➤   connection.query('SELECT * FROM accounts WHERE name= ?', [req.params.name],
➤       function(err, rows, fields) {
            if (err) {
                next(err);
                return;
            }
            res.send(JSON.stringify(rows));
        });
});
```

Prepared statements are by far the best solution against SQL injection attacks and are the favored approach. The second-best option is to use proper escaping, which is the method node-mysql uses currently. Whitelisting, while very effective, is the least favored. That's because it's time consuming to whitelist all possible endpoints and sometimes you need to have special characters in the query, which makes this approach less effective. Whichever method you use in your application, don't forget to limit the error messages.

Prepared like node-mysql

While you can use prepared statement syntax with node-mysql, such syntax is internally executed using connection.escape() and is not prepared statements. node-mysql2[8] is a library that does support prepared statements, and hopefully they will soon be available in node-mysql as well.

8. https://github.com/sidorares/node-mysql2

Now that you know the three ways to protect yourself when constructing database commands, you can feel confident that you aren't such an easy target for attackers. You might be able to order that Tesla after all.

Watch Out for Sneaky Issues

Before you start patting yourself on the back for a job well done, you still have a few hurdles left to address. Developers often use an ORM (object-relational mapper) instead of constructing commands manually, and that can introduce some unexpected behavior.

Numerous ORMs are available for Node.js and various databases. Let's look at one of the popular ORM mappers for MySQL, MariaDB, SQLite, and PostgreSQL in Node.js—Sequelize.[9] While ORMs typically implement internal escaping based on model properties and types, Sequelize does not always perform thorough input cleaning. Some inputs are left vulnerable and can be used to construct malicious SQL statements. It would be foolish to assume the ORM is going to do something without checking. Trust but verify.

ORMs by their nature introduce overhead into your application, because they construct interfaces around your data structures. This can lead to serious performance issues in some cases. So using an ORM isn't always the best solution. But if you're going to use ORMs, you need to test how they handle input cleaning. We will look at Sequelize.

First, let's do the setup:

chp-5-database/sequelize-example.js

```
'use strict';

var express = require('express');
var Sequelize = require('sequelize');
var args = require('minimist')(process.argv);

if(!args.u || !args.d || !args.p) {
    console.log('This example requires the ' +
    '-u (user), ' +
    '-d (mysql db) and ' +
    '-p (password) command line variables');

    process.exit();
}

// Define connection to DB
var sequelize = new Sequelize(args.d, args.u, args.p, {
```

9. http://sequelizejs.com/

```
    dialect: 'mysql',
    port:    3306
});

// Define user model
var User = sequelize.define('user', {
    company: Sequelize.STRING,
    username: Sequelize.STRING
});
```

Now that we've done the database connection part, let's define the application paths:

chp-5-database/sequelize-example.js

```
var app = express();

app.get('/', function (req, res) {
    res.send('ok');
});

// Define a path where we can ask users by
// company name and optionally limit the response
app.get('/:company/:limit*?', function(req, res, next){

    console.log(req.params);
    User.findAll({
        where: {
            company: req.params.company
        },
        limit: req.params.limit || 0
    }).then(function(users) {
        res.send(JSON.stringify(users));
    }).catch(next);
});
```

Finally, we create our example database entries and set the ball rolling:

chp-5-database/sequelize-example.js

```
// Set up the database
sequelize
    .authenticate()
    .then(function() {
        // Sync the models
        return sequelize.sync({ force: true });
    })
    .then(function () {
        // Push example data into the database
        return User.bulkCreate([
            { username: 'karl', company: 'nodeswat' },
            { username: 'harri', company: 'nodeswat' },
            { username: 'jaanus', company: 'nodeswat' },
            { username: 'jaak', company: 'mektro' }
        ]).then(function() {
            // We are set up so start listening
            app.listen(3000);
        });
    })
    .catch(function (err) {
        console.log('Unable to connect to the database:', err)
        process.exit();
    });
```

Now when we send the URL /nodeswat we should get the following data dump:

```
[{
    "id": 1,
    "company": "nodeswat",
    "username": "karl",
    "createdAt": "2014-01-24T11:33:01.000Z",
    "updatedAt": "2014-01-24T11:33:01.000Z"
}, {
    "id": 2,
    "company": "nodeswat",
    "username": "harri",
    "createdAt": "2014-01-24T11:33:01.000Z",
    "updatedAt": "2014-01-24T11:33:01.000Z"
}, {
    "id": 3,
    "company": "nodeswat",
    "username": "jaanus",
    "createdAt": "2014-01-24T11:33:01.000Z",
    "updatedAt": "2014-01-24T11:33:01.000Z"
}]
```

Let's take a look at the actual database command that was executed. So far, so good:

```
SELECT * FROM `users` WHERE `users`.`company`='nodeswat';
```

When we add in malicious code as shown in the following URL, /nodeswat'";, we see that the quotation marks are properly escaped (as expected!):

```
SELECT * FROM `users` WHERE `users`.`company`='nodeswat\'\";';
```

When we use the limit parameter in the URL like /nodeswat/1;DROP%20TABLE%20users, we get an error. But let's look at the final command that was constructed. As you can see, this could have gone badly:

```
SELECT * FROM `users` WHERE `users`.`company`='nodeswat' LIMIT 1;DROP TABLE users;
```

The only reason the command didn't execute and drop the users table was that Sequelize uses node-mysql as its MySQL driver. As you saw earlier, node-mysql disables by default the ability to run multiple statements off a single command. But PostgreSQL's driver doesn't have that setting, so a similar attack will succeed on that database. You'll need to do validation or change the configuration if you're working with PostgreSQL or other databases that don't disable multiple queries by default:

```
var sequelize = new Sequelize(args.d, args.u, args.p, {
    dialect: 'postgres',
    port: 5432
});
```

Here, the solution is to either not allow the user to set the limit parameter or validate that you're dealing with a number.

The moral of this example is that you must be ever vigilant when using third-party modules and talking to the database. You should always limit user interaction with your database and test what users are allowed to do. Knowing this makes you security conscious and in a better position to write secure applications.

Mitigate Injection Attacks in NoSQL Databases

MongoDB and CouchDB[10] are widely used alternatives to relational databases when building Node.js applications. They don't use a query language like SQL for mapping the data—hence the name NoSQL. Instead they have their own methods and queries. You may now think that using NoSQL means SQL injection is not a problem for you. Alas, that isn't quite true.

Strictly speaking, SQL injection doesn't affect NoSQL databases. The thing is, SQL injection isn't the only form of database injection, and there are other ways to inject commands despite not using the traditional SQL syntax. These NoSQL injection attacks execute within a procedural language rather than in the declarative SQL language, so the potential impact of these attacks is greater.

In the following examples we'll be using MongoDB,[11] since it's the most popular NoSQL database in use with Node.js.

The first security issue for NoSQL databases is that by default they don't have any authentication. Instead, they filter connections only to localhost. As you saw in *Start with the Basics: Set Up the Database*, on page 6, that's not necessarily a good thing.

10. http://couchdb.apache.org/
11. http://www.mongodb.org

Let’s see how to connect to our configured MongoDB using a password with one of the most (if not the most) popular MongoDB ORMs in use—Mongoose:[12]

```
mongoose.connect('mongodb://user:pass@localhost:port/database');
```

The following example uses `mongoose` and `express` to illustrate how MongoDB can be susceptible to attack:

chp-5-database/mongoose-example.js

```
'use strict';

var express = require('express');
var mongoose = require('mongoose');
var args = require('minimist')(process.argv);

if(!args.d) {
    console.log('This example requires the -d (mongoose db) command line variable');
    process.exit();
}

// Connect to mongoose db
mongoose.connect(args.d);
mongoose.connection.on('error', function (err) {
    console.error('connection error:' + err);
    process.exit();
});

// Define user model
var userSchema = new mongoose.Schema({
    username:  { type: String, required: true, index: { unique: true } },
    company: { type: String, required: true },
    age: { type: Number, required: true}
});
var User = mongoose.model('User', userSchema);

User.remove().exec(); // Delete all previous Users.

var app = express();

app.get('/', function(req, res){
    res.send('ok');
});

app.get('/:age', function (req, res, next) {
    // Lets implement a completely unvalidated way to query the documents
    User.find({ $where: 'this.age < ' + req.params.age }, function (err, users) {
        if(err) {
            next(err);
```

12. http://mongoosejs.com/

```
            return;
        }
        res.send(JSON.stringify(users));
    });
});
// Fill database
User.create([
    { username: 'karl', company: 'nodeswat', age: 25 },
    { username: 'harri', company: 'nodeswat', age: 35 },
    { username: 'jaanus', company: 'nodeswat', age: 45 },
    { username: 'jaak', company: 'mektro', age: 55 }
], function (err) {
    if(err) {
        console.error(err);
        process.exit(1);
    }
    console.log('Listening');
    app.listen(3000);
});
```

With this application, you can ask for a list of users under a specific age. Constructing a URL with /40 would result in a data dump of some users:

```
[{
    "username": "karl",
    "company": "nodeswat",
    "age": 25,
    "_id": "52e25cc4251a7ce88b050e75",
    "__v": 0
}, {
    "username": "harri",
    "company": "nodeswat",
    "age": 35,
    "_id": "52e25cc4251a7ce88b050e76",
    "__v": 0
}]
```

The user input was not validated before constructing the search, which means attackers can add malicious code into the statement. MongoDB won't be affected by SQL statements, but it does support JavaScript commands in its queries. The attacker can execute JavaScript statements on the database layer. It may look something like this URL, which would trigger a ten-second loop in the database:

```
/40;var%20date=new%20Date();%20do%7BcurDate%20=%20new%20Date();%7Dwhile(curDate-date<10000);
```

Keep in mind that this is a simple example; the attacker has access to the whole JavaScript syntax to craft a more complicated query. Because most

NoSQL databases don't support prepared statements, you're left with two solutions—validation and escaping. Be sure to use them liberally.

You just saw that NoSQL is not inherently safer just because it does not have SQL. When constructing complex queries with user input, make sure the data falls within the narrowly defined parameters of your query, just as you would with a SQL database.

Wrapping Up

Databases are an integral and powerful part of a web application, and you must secure all transactions in order to protect your clients' data. As you learned in this chapter, you must secure your database connections and limit access privileges where you can. You must also be vigilant about escaping and validating all user input that comes into contact with the database, even if it's a NoSQL database. Implementing these two steps will greatly increase the security of your data.

Now that you've secured how the application communicates with the database, the attackers will find it harder to target your application. Don't get too cozy just yet, because there are many more attack vectors to defeat out there. Next we'll move on to another common issue in web application design that also affects databases: concurrency. Let's go.