6 Persisting to Databases

This chapter covers the following recipes

- Connecting and sending SQL to a MySQL server
- Connecting and sending SQL to a Postgres server
- Storing and Retrieving Data with MongoDB
- Storing and Retrieving Data with Redis
- Embedded Persistance with LevelDB

Introduction

In many cases, relational databases became a de facto standard for nearly all data scenarios. This led to the necessity of imposing relationships on otherwise loosely-related data (such as website content) in an attempt to squeeze it into our relational mental model.

In recent times, though, there has been a movement away from relational databases towards NoSQL, a non-relational paradigm; the driving force being the fact that we tailor our technology to best suit our data, rather than trying to fit our data according to our technology.

In this chapter, we will look at various data storage technologies with examples of their usage in Node.

Connecting and sending SQL to a MySQL server

Structured Query Language has been a standard since 1986, and it's the prevailing language for relational databases. MySQL is the most popular SQL relational database server around, often appearing in the prevalent Linux Apache MySQL PHP (LAMP) stack.

If a relational database was conceptually relevant to our goals in a new project, or we were migrating a MySQL-backed project from another framework to Node, the `mysql` module would be particularly useful.

In this task, we will discover how to connect to a MySQL server with Node and execute SQL queries across the wire.

Getting Ready

We'll need a MySQL server to connect to. By default, the `mysql` client module connects to localhost, so we'll have MySQL running locally

On Linux we can see if MySQL is already installed with the following command:

```
$ whereis mysql
```

On macOS we can use

```
$ type -a mysql
```

On Windows we can use the GUI and check the package manager via the control panel.

We can see if the mysql server is running using the following command:

```
$ mysqladmin -u root ping
```

If it is installed but not running, we can use the following command on Linux:

```
$ sudo service mysql start
```

Or on MacOS:

```
$ mysql.server start
```

If MySQL isn't installed, we can use the relevant package manager for our system (homebrew, apt-get/synaptic, yum, and so on) and follow instructions to start the server.

If we're using Node on Windows, we can head to http://dev.mysql.com/downloads/mysql and download the installer.

MariaDB

This recipe, including extra code in the **There's More** section will run with MariaDB without changing any code. We can install MariaDB instead of MySQL if we choose. The advantages of MariaDB is it's owned by an open source foundation (MariaDB Foundation) instead of a corporation (Oracle) and it has dynamic storage options (instead of requiring recompilation). See http://mariadb.com for more.

Once we have MySQL up and running, let's create a folder called `mysql-app` with an `index.js` file.

Then, we'll initialize the folder with a `package.json` file and grab the `mysql` driver module, which is a pure JavaScript module (as opposed to a C++ binding to the MySQL C driver).

```
$ npm init -y
$ npm install --save mysql
```

How to do it

In `mysql-app/index.js` let's require the `mysql` module and open a connection to our locally running MySQL instance:

```javascript
const mysql = require('mysql')
const db = mysql.createConnection({
  user: 'root',
  //password: 'pw-if-set',
  //debug: true
})
```

We need a database to connect to. Let's keep things interesting and make a quotes database. We can do that by passing SQL to the query method as follows:

```javascript
db.query('CREATE DATABASE quotes')
db.query('USE quotes')
```

Now, we'll create a table with the same name:

```javascript
db.query(
```

```
    'CREATE TABLE quotes.quotes (' +
    'id INT NOT NULL AUTO_INCREMENT, ' +
    'author VARCHAR( 128 ) NOT NULL, ' +
    'quote TEXT NOT NULL, PRIMARY KEY ( id )' +
    ')'
)
```

If we were to run our code in it's current state more than once, we would notice that the program fails with an unhandled exception.

The MySQL server is sending an error to our process, which is then throwing that error. The source of the error is down to the `quotes` database already existing on the second run.

We want our code to be versatile enough to create a database if necessary, but not to throw an error if it's not there.

We can prevent the process from crashing by listening for an `error` event on the `db`. We'll attach a handler that will throw any errors that don't relate to pre-existing tables:

```
const ignore = new Set([
  'ER_DB_CREATE_EXISTS',
  'ER_TABLE_EXISTS_ERROR'
])

db.on('error', (err) => {
  if (ignore.has(err.code)) return
  throw err
})
```

Finally, we'll insert our first quote into the table and send a `COM_QUIT` packet (using `db.end`) to the MySQL server.

This will only close the connection once all the queued SQL code has been executed.

```
db.query(`
  INSERT INTO quotes.quotes (author, quote)
  VALUES ("Bjarne Stroustrup", "Proof by analogy is fraud.");
`)

db.end()
```

We can verify our program by running it:

```
$ node index.js
```

Then in another terminal executing the following:

```
$ mysql -u root -D quotes -e "select * from quotes;"
```

If we run our program more than once, the quote will be added several times.

How it works

The `createConnection` method establishes a connection to the server and returns a `db` instance for us to interact with.

We can pass in an options object that may contain an assortment of various properties. We have included `password` and `debug` properties, though commented out as they're not needed in the common case.

If we uncomment `debug`, we can see the raw data being sent to and from the server. We only need uncomment `password` if our MySQL server has a password set.

The `mysql` module API ✍

Check out the `mysql` module's GitHub page for a list of all the possible options at https://github.com/felixge/node-mysql.

The `db.query` call sends SQL to the MySQL server, which is then executed.

When executed, the SQL creates a database named "quotes" (using `CREATE` and `DATABASE`) and a `TABLE` also named quotes.

We then insert our first record (using `INSERT`) into our database.

When used without a callback, the `db.query` method queues each piece of SQL passed to it, executing statements asynchronously (preventing any blocking of event loop), but sequentially within the SQL statement queue.

When we call `db.end`, the connection closing task is added to the end of the queue.

If we wanted to disregard the statement queue and immediately close the connection, we could use `db.destroy`.

Our `ignore` set holds MySQL error codes `ER_DB_CREATE_EXISTS` and `ER_TABLE_EXISTS_ERROR`. We check the `ignore` set using the `has` method when the `error` event fires on the `db` object. If there's a match we simply return early from the `error` event handler, otherwise we `throw` the error.

There's more

SQL queries are often generated from user input, but this can be open to exploitation if precautions aren't taken. Let's look at cleaning user input, and also find out how to retrieve data from a MySQL database.

Avoiding SQL Injection

As with the other languages that build SQL statements with string concatenation, we must prevent the possibilities of SQL injection attacks to keep our server safe. Essentially, we must clean (that is, escape) any user input to eradicate the potential for unwanted SQL manipulation.

Let's copy the `mysql-app` folder and name insert-quotes`.

To implement the concept of user input in a simple way, we'll pull the arguments from the command line, but the principles and methods of data cleaning extend to any input method (for example, via a query string on request).

Our basic CLI API will look like this:

```
$ node index.js "Author Name" "Quote Text Here"
```

Quotation marks are essential to divide the command-line arguments, but for the sake of brevity, we won't be implementing any validation checks.

> ### Command-line parsing with minimist 💡
>
> For more advanced command-line functionality, check out the excellent `minimist` module, http://npm.im/minimist

To receive an author and quote, we'll load the two command line arguments into a new `params` object:

```
const params = {
  author: process.argv[2],
  quote: process.argv[3]
}
```

Our first argument is at index 2 in the `process.argv` array because `process.argv` includes all command line arguments (the name of the binary (node) and the path being executed).

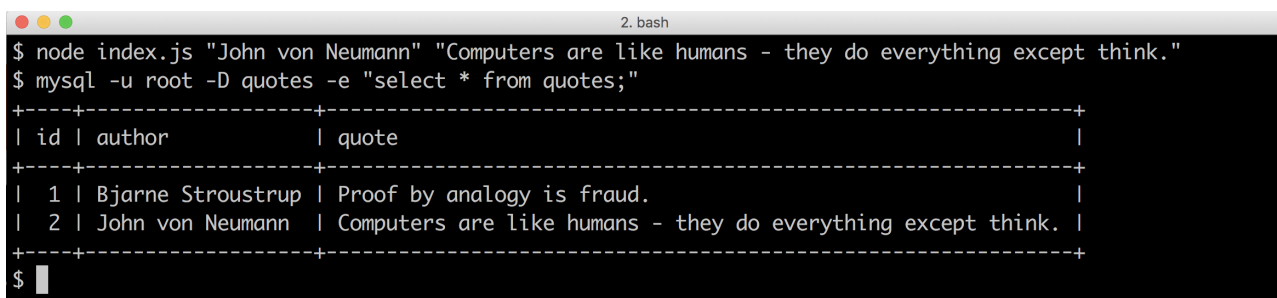Now, let's slightly modify our `INSERT` statement passed to `db.query`:

```
if (params.author && params.quote) {
  db.query(`
    INSERT INTO quotes.quotes (author, quote)
    VALUES (?, ?);
  `, [params.author, params.quote])
}
```

The `mysql` module can seamlessly clean user input for us. We simply use the question mark (?) as a placeholder and then pass our values (in order) as an array to the second parameter of `db.query`.

Let's try it out:

```
$ node index.js "John von Neumann" "Computers are like humans — they do
$ mysql -u root -D quotes -e "select * from quotes;"
```

This should give something like the following figure:


```
2. bash
$ node index.js "John von Neumann" "Computers are like humans - they do everything except think."
$ mysql -u root -D quotes -e "select * from quotes;"
+----+-----------------+-------------------------------------------------------------------+
| id | author          | quote                                                             |
+----+-----------------+-------------------------------------------------------------------+
|  1 | Bjarne Stroustrup | Proof by analogy is fraud.                                       |
|  2 | John von Neumann  | Computers are like humans - they do everything except think.    |
+----+-----------------+-------------------------------------------------------------------+
$
```

Inserting a record to MySQL via Node

Querying a MySQL Databases

Let's copy the `insert-quotes` folder from the previous section, and save it as `quotes-app`.

We'll extend our app further by outputting all the quotes for an author, irrespective of whether a quote is provided.

Let's add the following code just above the final `db.end` call:

```
if (params.author) {

  db.query(`
    SELECT * FROM quotes
    WHERE author LIKE ${db.escape(params.author)}
  `).on('result', ({author, quote}) => {
    console.log(`${author} ${quote}`)
  })

}
```

On this occasion, we've used an alternative approach to clean user input with `db.escape`. This has exactly the same effect as the former, but only escapes a single input. Generally, if there's more than one variable, the former method would be preferred.

The results of a `SELECT` statement can be accessed either by passing a callback function or by listening for the `result` event.

We can safely call `db.end` without placing it in the end event of our SELECT query because `db.end` only terminates the connection when all the queries are done.

We can test our addition like so:

```
$ node index.js "Bjarne Stroustrup"
```

We can use the SQL wildcard (`%`) to get all quotes:

```
$ node index.js "%"
```

See also

- TBD

Connecting and sending SQL to a Postgres

server

Postgres is a object-relational databases. It gives us everything that MySQL does, along with enhanced commands, and the ability to store and query object data. This allows us to use the same database for for both relational and document type data.

In this recipe we're going to implement the same quotes application as we did in the previous recipe. In the **There's More** section we'll explore Postgres' additional object storage potential.

Getting Ready

We'll need to install a Postgres server.

On macOS we can use homebrew (http://brew.sh):

```
$ brew install postgres
```

For Windows systems, we can download a GUI installer from https://www.postgresql.org/download/windows/.

For Linux systems we can obtain an appropriate package from https://www.postgresql.org/download/linux/.

Once installed (and started) we'll want to create a database named after our system username.

After installation, in the usual command terminal run:

```
$ createdb `whoami`
```

Once we have Postgres up and running, let's create a folder called `postgres-app` with an `index.js` file.

Then, we'll initialize the folder with a `package.json` file and grab the `pg` module, which is a pure JavaScript module (as opposed to a C++ binding to the MySQL C driver).

```
$ npm init -y
```

```
$ npm install --save pg
```

How to do it

Let's begin by requiring the `pg` module and create a new client:

```
const pg = require('pg')
const db = new pg.Client()
```

We don't need to provide any configuration, because Postgres adds environment variables which the `pg` module reads from.

Now, let's add a `params` object which will hold user supplied `author` and `quote`:

```
const params = {
  author: process.argv[2],
  quote: process.argv[3]
}
```

Next we'll connect to the database, conditionally create a table to store our quotes and insert a quote when the user supplies both author and quote arguments.

```
db.connect((err) => {
  if (err) throw err
  db.query(`
    CREATE TABLE IF NOT EXISTS quotes (
      id SERIAL,
      author VARCHAR ( 128 ) NOT NULL,
      quote TEXT NOT NULL, PRIMARY KEY ( id )
    )
  `, (err) => {
    if (err) throw err

    if (params.author && params.quote) {
      db.query(`
        INSERT INTO quotes (author, quote)
        VALUES ($1, $2);
      `, [params.author, params.quote], (err) => {
        if (err) throw err
        list(db, params)
      })
    }

    if (!params.quote) list(db, params)
```

```
    })
  })
```

Finally we'll implement the `list` function that's called after the insertion, or in cases where only the author is supplied.

```
function list (db, params) {
  if (!params.author) return db.end()
  db.query(`
    SELECT * FROM quotes
    WHERE author LIKE ${db.escapeLiteral(params.author)}
  `, (err, results) => {
    if (err) throw err
    results.rows.forEach(({author, quote}) => {
      console.log(`${author} ${quote}`)
    })
    db.end()
  })
}
```

Now we should be able to try out our app with:

```
$ node index.js "Neal Stephenson" "To condense fact from the vapor of n
```

This will output any quotes by Neal Stephenson, including the one we just added.

How it works

Postgres offers slight SQL variants (along with a non-standard superset) SQL compared to MySQL.

Accepted convention in Postgres is to expect a database named after the user account that owns a gives process. This is why in the getting ready section we ran `createdb `whoami`` on the command line.

When we instantiate `pg.Client` (storing the instance to the `db` variable) it reads the `USER` (or `USERNAME` on Windows) environment variable and attempts to connect to a database named after the current user, as that user, on the default Postgres port. By default the database is passwordless.

The upshot is we didn't need to supply a configuration to `pg.Client` .

We call `pg.connect` to connect to the Postgres server and use the database (as

named after the user). This will trigger the callback we supplied. Since we're writing a command line application we will throw any errors that come through any callbacks for instant user feedback.

From this point we're simply sending SQL to the Postgres server via the `db.query` method. If the user has supplied both author and quote arguments we perform an `INSERT`. We clean the user input by using placeholders (`VALUES($1, $2)`) and supplying an array of values matching the placeholder index.

The `list` function sends a `SELECT` query, this time we clean user input using `db.escapeLiteral`.

There's More

Postgres is a hybrid object-relational database, we can store and query both relational and object data, whilst the `pg` module can interface with Postgres with both pure JavaScript and C bindings.

Using native bindings

The `pg` module primarily provides a pure JavaScript driver for Postgres, but can also supply a consistent API over a native C driver with Node bindings (the `pg-native` module) which should provide enhanced performance.

Let's copy our `postgres-app` folder to `postgres-native-app`, then install the `pg-native` module:

```
$ cp -fr postgres-app postgres-native-app
$ cd postgres-native-app
$ npm install --save-opt pg-native
```

When we install a module with `--save-opt` it is added to the `package.json` file as an optional dependency (in the `optionalDependencies` field).

This means if we install `pg-native` on a machine that fails to compile, the installation process will still report successful completion.

The first line of `index.js` we change to:

```
const pg = require('pg').native || require('pg')
```

If `pg-native` failed to install, the `native` property of the `pg` module will be `null`, in which case we drop back to the JavaScript implementation.

Finally we need to alter the `list` function slightly:

```javascript
function list (db, params) {
  if (!params.author) return db.end()
  db.query(`
    SELECT * FROM quotes
    WHERE author LIKE $1
  `, [params.author], (err, results) => {
    if (err) throw err
    results.rows.forEach(({author, quote}) => {
      console.log(`${author} ${quote}`)
    })
    db.end()
  })
}
```

The native API is high parity with the pure JavaScript API, though with very slightly reduced functionality. This means we can't use the `db.escapeLiteral` function, so revert to the (more normative) placeholder injection (`LIKE $1`).

Storing Object-modelled Data

Postgres also has the ability to store object data, allowing for hybrid data strategies with the same database. In other words, we can couple a relational approach with a noSQL document store paradigm in the same database.

Let's convert our main recipe to an object-relational approach instead of pure relational.

Let's copy the `postgres-app` folder from our main recipe to `postgres-object-app`.

```
$ cp postgres-app postgres-object-app
```

Now let's edit the `index.js` file.

We'll start by changing the `db.connect` call to the following:

```javascript
db.connect((err) => {
  if (err) throw err
```

```
    db.query(`
      CREATE TABLE IF NOT EXISTS quote_docs (
        id SERIAL,
        doc jsonb,
        CONSTRAINT author CHECK (length(doc->>'author') > 0 AND (doc->>'a
        CONSTRAINT quote CHECK (length(doc->>'quote') > 0 AND (doc->>'quo
      )
    `, (err) => {
      if (err) throw err

      if (params.author && params.quote) {
        db.query(`
          INSERT INTO quote_docs (doc)
          VALUES ($1);
        `, [params], (err) => {
          if (err) throw err
          list(db, params)
        })
      }

      if (!params.quote) list(db, params)

    })
  })
```

The only thing we've changed here is the SQL queries (and their inputs).

We've altered the name of the table we create (from `quotes` to `quote_docs`), and we define two fields instead of three. The `id` field remains the same, but instead of `author` and `quote` fields we have a `doc` field with type `jsonb`. Postgres has two object types, `json` and `jsonb` (JSON Binary). The `json` datatype is little more than a text field with JSON validation, whereas the `jsonb` is structured, allowing for queries and even index creation within objects. We might use the `json` type if all we're interested in is storing data blobs wholesale (such as log storage), but `jsonb` if we want to interact with the data inside the database.

We also added `CONSTRAINT` statements to validate the objects passed to Postgres.

In the second query, we only need to pass in one value, an object with `author` and `quotes` properties. So in our inputs array (second argument to the second occurence `db.query`), we simply pass the `params` object. Postgres takes this object, converts it to JSON, and then stores it as the `jsonb` datatype.

Now let's modify the `list` function to query the object data:

```
function list (db, params) {
  if (!params.author) return db.end()
  db.query(`
    SELECT * FROM quote_docs
    WHERE doc ->> 'author' LIKE ${db.escapeLiteral(params.author)}
  `, (err, results) => {
    if (err) throw err
    results.rows
      .map(({doc}) => doc)
      .forEach(({author, quote}) => {
        console.log(`${author} ${quote}`)
      })
    db.end()
  })
}
```

We've modified the SQL in our `list` function to run a nested query, `doc` is the `jsonb` datatype, so we use the `->>` operator to query keys within the objects found.

> **The JSONB datatype**
>
> We're only scratching the surface here, for see
> https://www.postgresql.org/docs/9.4/static/functions-json.html for more
> information on Postgres JSONB query operators.

See also

- TBD

Storing and Retrieving Data with MongoDB

MongoDB is a NoSQL database offering that maintains a philosophy of performance over features. It's designed for speed and scalability. Instead of working relationally, it implements

a document-based model that has no need for schemas (column definitions). The document model works well for scenarios where the relationships between data are flexible and where minimal potential data loss is an acceptable cost for speed enhancements (a blog, for instance).

While it is in the NoSQL family, MongoDB attempts to sit between two worlds,

providing a syntax reminiscent of SQL but operating non-relationally.

In this task, we'll implement the same quotes database as in the previous recipe, using MongoDB instead of MySQL.

Getting Ready

We want to run a MongoDB server locally. It can be downloaded from http://www.mongodb.org/download-center (we may also be able to install it with our OS's package manager).

Once installed, let's start the MongoDB service, `mongod`, in the default debug mode:

```
$ mkdir ./data
$ mongod --dbpath ./data
```

Where `./data` is a folder that holds the database files.

This allows us to observe the activities of `mongod` as it interacts with our code.

> ### Managing the MongoDB Service 🖎
>
> More information on starting and correctly stopping MongoDB can be found at https://docs.mongodb.com/manual/tutorial/manage-mongodb-processes/.

Now (in a new terminal) let's create a new folder called `mongo-app` with an `index.js` file.

To interact with MongoDB from Node, we'll need to install the `mongodb` native binding's driver module (in `mongo-app`):

```
$ npm init -y
$ npm install --save mongodb
```

How to do it

Let's `require` the `mongodb` driver, and create an instance of the `MongoClient` constructor supplied via the `mongodb` object:

```
const {MongoClient} = require('mongodb')
const client = new MongoClient()
```

We used object destructuring to pull the `MongoClient` constructor from the `mongodb` exported object.

To receive an author and quote, we'll load the two command line arguments into a new `params` object:

```
const params = {
  author: process.argv[2],
  quote: process.argv[3]
}
```

Now, we connect to our quotes database and load (or create, if necessary) our quotes collection (a table would be the closest similar concept in SQL):

```
client.connect('mongodb://localhost:27017/quotes', ready)

function ready (err, db) {
  if (err) throw err
  const collection = db.collection('quotes')
  db.close()
}
```

Port 27017 is the default port assigned to a `mongod` service. This can be modified when we start a `mongod` by passing a `--port` flag.

Now let's expand our `ready` function to insert a new document (in SQL terms, this would be a record) when the user supplies both an author and quote:

```
function ready (err, db) {
  if (err) throw err
  const collection = db.collection('quotes')
  if (params.author && params.quote) {
    collection.insert({
      author: params.author,
      quote: params.quote
    }, (err) => {
      if (err) throw err
    })
  }
```

```
    db.close()
  }
```

Finally we'll expand `ready` one more time, so that it outputs a list of quotes according to supplied author. Our final `ready` function should look like so:

```
function ready (err, db) {
  if (err) throw err
  const collection = db.collection('quotes')

  if (params.author && params.quote) {
    collection.insert({
      author: params.author,
      quote: params.quote
    }, (err) => {
      if (err) throw err
    })
  }

  if (params.author) {
    collection.find({
      author: params.author
    }).each((err, doc) => {
      if (err) throw err
      if (!doc) {
        db.close()
        return
      }
      console.log('%s: %s \n', doc.author, doc.quote)
    })
    return
  }

  db.close()
}
```

Now we can run our app:

```
$ node index.js "Woody Allen" "I'd call him a sadistic hippophilic necr
```

This will immediately output our entered quote (and any other quotes by Woody Allen if they we're previously entered).

How it works

When we call `client.connect`, we pass in a URI with the `mongodb://` protocol as the first parameter. The `mongodb` module will parse this string and attempt to connect to the specified `quotes` database. MongoDB will intelligently create this database if it doesn't exist, so unlike MySQL, we don't have to plaster over awkward errors.

Once the connection is made, our `ready` callback function is executed where we can interact with the database via the `db` parameter.

We start off by grabbing our `quotes` collection using `db.collection`. A collection is similar to an SQL table which holds all our database fields. However, rather than the field values being grouped by columns, a collection contains multiple documents (such as records) where each field holds both the field name and its value (the documents are very much like JavaScript objects).

If both quote and author have been passed as arguments, we invoke the `collection.insert` method, passing in an object as our document.

Finally, we use `collection.find`, which is comparable to the `SELECT` SQL command, passing in an object that specifies the author field and its desired value. The `mongodb` driver module provides a convenience method (`each`) that can be called on the result of the `collection.find` method. The `each` method executes the iterator function passed to it for each document as and when it's found. The `doc` parameter is set to `null` for the last call of the iterator function, signalling that MongoDB has returned all the records.

So we check if `doc` is falsy (it should always be an object or `null` but just in case, checking for `!doc` means we cover `false` and `undefined` as well), and then call `db.close` returning early from the function. If `doc` isn't falsy we proceed to log out the author and quote.

The second and final `db.close` call situated at the end of the `ready` function is invoked only when there are no arguments defined via the command line.

There's more

Let's check out some other useful MongoDB features.

Indexing and Aggregation

Indexing causes MongoDB to create a list of values from a chosen field. Indexed

fields accelerate query speeds because a smaller set of data can be used to cross-reference and pull from a larger set. We can apply an index to the author field and see performance benefits, especially as our data grows. Additionally, MongoDB has various commands that allow us to aggregate our data. We can group, count, and return distinct values.

Let's create and output a list of authors found in our database.

We'll create a file in same `mongo-app` folder, called `author.js`.

```javascript
const {MongoClient} = require('mongodb')
const client = new MongoClient()

client.connect('mongodb://localhost:27018/quotes', ready)

function ready (err, db) {
  if (err) throw err
  const collection = db.collection('quotes')
  collection.ensureIndex('author', (err) => {
    if (err) throw err
    collection.distinct('author', (err, result) => {
      if (err) throw err
      console.log(result.join('\n'));
      db.close()
    })
  })
}
```

As usual, we opened up a connection to our quotes database, grabbing our quotes collection. Using `collection.ensureIndex` creates an index only if one doesn't already exist.

Inside the callback, we invoke the `collection.distinct` method, passing in `author`. The result parameter in our callback function is an array which we join (using the `join` method) to a newline delimited string and output the result to the terminal.

Updating modifiers, `sort` and `limit`

We can make it possible for a hypothetical user to indicate if they were inspired by a quote and then use the `sort` and `limit` commands to output the top ten most inspiring quotes.

In reality, this would be implemented with some kind of user interface (for example,

in a browser), but we'll again emulate user interactions using the command line.

Let's create a new file named `votes.js`.

First, in order to vote for a quote, we'll need to reference it. This can be achieved with the unique `_id` property.

Let's write the following code:

```
const {MongoClient, ObjectID} = require('mongodb')
const client = new MongoClient()
const params = {id: process.argv[2]}

client.connect('mongodb://localhost:27017/quotes', ready)

function ready (err, db) {
  if (err) throw err
  const collection = db.collection('quotes')

  if (!params.id) {
    showIds(collection, db)
    return
  }

  vote(params.id, db, collection)
}
```

If an argument is supplied it's loaded into `params.id`, if `params.id` is empty, then we'll print out the ID of each quote in our collection.

Our `ready` function calls a `showIds` function to achieve this, let's write the `showIds` function:

```
function showIds (collection, db) {
  collection.find().each((err, doc) => {
    if (err) throw err
    if (doc) {
      console.log(doc._id, doc.quote)
      return
    }
    db.close()
  })
}
```

Now let's do our vote handling by creating our `vote` function like so:

```
function vote (id, db, collection) {
  const query = {
    _id  : ObjectID(id)
  }
  const action = {$inc: {votes: 1}}
  const opts = {safe: true}
  collection.update(query, action, opts, (err) => {
    if (err) throw err
    console.log('1 vote added to %s by %s', params.id)
    const by = {votes: 1}
    const max = 10
    collection.find().sort(by).limit(max).each((err, doc) => {
        if (err) throw err
        if (doc) {
          const votes = doc.votes || 0
          console.log(`${votes} | ${doc.author}: ${doc.quote.substr(0,
          return
        }
        db.close()
    })
  })
  return
}
```

To use we first run `votes.js` without any arguments:

```
$ node votes.js
```

This will output a list of MongoDB ID's along side the quote. We can pick one of the ID's and run our `votes.js` script again, but this time passing the ID:

```
$ node votes.js 586eacd7f959a401fa63acc2
```

This will then output the amount of votes (including the recent vote), each quote has received.

MongoDB IDs must be encoded as a Binary JSON (BSON) ObjectID. Otherwise, the update command will look for a string, failing to find it. So, we convert `id` into an ObjectID using the `mongodb` `ObjectID` function (which we destructure from the `mongodb` module on the first line of `votes.js`).

The `$inc` property is a MongoDB modifier that performs the incrementing action inside the MongoDB server, essentially allowing us to outsource the calculation. To

use it, we pass a document (object) alongside it containing the key to increment and the amount to increase it by. So our `action` object essentially instructs MongoDB to increase the `votes` key by one.

The `$inc` modifier will create the `votes` field if it doesn't exist and increment it by one (we can also decrement using minus figures).

Our `opts` object has a `safe` property set to `true`, this ensures that the callback isn't fired until MongoDB has absolutely confirmed the update was written (although it will make operations slower).

Inside the `update` callback, we execute a chain of methods (`find`, `sort`, `limit`, `each`). A call to `find`, without any parameters, returns every document in a collection. The `sort` method requires an object whose properties match the keys in our collection. The value of each property can either be `-1` or `+1`, which specifies ascending and descending order respectively.

The `limit` method takes accepts a number representing the maximum amount of records to return and the `each` method loops through all our records. Inside the `each` callback, we output vote counts alongside each author and quote. When there are no documents remaining, the `each` method will call the callback on final time, setting `doc` to `null` - in this case we fall through both `if` statements to the final `db.close` call.

See also

- TBD

Storing and Retrieving Data with Redis

Redis is a key value store which functions in operational memory with blazingly fast performance.

Redis is excellent for certain tasks, as long as the data model is fairly simple.

Good examples of where Redis shines are in site analytics, server-side session cookies, and providing a list of logged-in users in real time.

In the spirit of this chapters theme, let's reimplement our quotes database with Redis.

Getting Ready

Let's create a new folder called `redis-app`, with an `index.js` file, initialize it and install `redis`, `steed` and `uuid`.

```
$ mkdir redis-app
$ cd redis-app
$ touch index.js
$ npm init -y
$ npm install --save redis steed uuid
```

We also need to install the Redis server, which can be downloaded from http://www.redis.io/download along with the installation instructions.

How to do it

Let's load supporting dependencies along with the `redis` module, establish a connection, and listen for the ready event emitted by the client.

We'll also load the command-line arguments into the `params` object.

Let's kick off `index.js` with the following code:

```javascript
const uuid = require('uuid')
const steed = require('steed')()
const redis = require('redis')
const client = redis.createClient()
const params = {
  author: process.argv[2],
  quote: process.argv[3]
}
```

Now we'll lay down some structure. We're going to check whether an author and/or quote has been provided via the command line. If only author was specified we'll list out all quotes for that author.
If both have been supplied, we'll add the quote to our Redis store, and then list out the quotes as well. If no arguments were given, we'll just close the connection to Redis, allowing the Node process to exit.

Let's add the following to `index.js`:

```javascript
if (params.author && params.quote) {
```

```
    add(params)
    list((err) => {
      if (err) console.error(err)
      client.quit()
    })
    return
  }

  if (params.author) {
    list((err) => {
      if (err) console.error(err)
      client.quit()
    })
    return
  }

  client.quit()
```

Now we'll write the `add` function:

```
function add ({author, quote}) {
  const key = `Quotes: ${Math.random().toString(32).replace('.', '')}`
  client.hmset(key, {author, quote})
  client.sadd(`Author: ${params.author}`, key)
}
```

Finally, we'll write the `list` function:

```
function list (cb) {
  client.smembers(`Author: ${params.author}`, (err, keys) => {
    if (err) return cb(err)
    steed.each(keys, (key, next) => {
      client.hgetall(key, (err, {author, quote}) => {
        if (err) return next(err)
        console.log(`${author} ${quote} \n`)
        next()
      })
    }, cb)
  })
}
```

We should now be able to add a quote (and see the resulting stored quotes):

```
$ node index.js "Steve Jobs" "Stay hungry, stay foolish."
```

This should output the quote we just added:

```
Steve Jobs Stay hungry, stay foolish.
```

How it works

If both author and quote are specified via the command line, we go ahead and generate a random key prefixed with `Quote:`. So, each key will look something like `Quote:0e3h6euk01vo`. It's a common convention to prefix the Redis keys with names delimited by a colon, as this helps us to identify keys when debugging.

We pass our key into `client.hmset`, a wrapper for the Redis `HMSET` command, which allows us to create multiple hashes.

Essentially when called with `params.author` set to "Steve Jobs" and `params.quote` set to "Stay hungry, stay foolish." the following command is being sent to Redis and executed (where `HASH` is the string we generate with `Math.random().toString(32).replace('.', '')`):

```
HMSET Quotes:HASH author "Steve Jobs" quote "Stay hungry, stay foolish.
```

Every time we store a new quote with `client.hmset`, we add the `key` for that quote to the relevant author set via the second parameter of `client.sadd`.

The `client.sadd` method allows us to add a member to a Redis set (a set is like an array of strings). The key for our `SADD` command is based on the intended author. So, in the preceding Steve Jobs quote, the key to pass into `client.sadd` would be `Author:Steve Jobs`.

In our `list` function we execute the `SMEMBERS` command using `client.smembers`. This returns all the values we stored to a specific author's set. Each value in an author's set is a key for a quote related to that author.

We use `steed.each` to loop through the keys, executing `client.hgetall` in parallel. Redis `HGETALL` returns a hash (which the `redis` module converts to a JavaScript object). This hash matches the object we passed into `client.hmset` in the `add` function.

Each author and quote is then logged to the console, and the final `cb` argument passed to `steed.each` is called. This in calls the callback function passed to

`list` in both `if` statements. Here we check for any errors and call `client.quit` to gracefully close the connection.

There's more

Redis is a speed freak's dream, but we can still be faster. Let's also take a brief look at authenticating with a Redis server.

Command Batching

Redis can receive multiple commands at once. The `redis` module has a `multi` method, which sends commands in batch.

Let's copy the `redis-app` folder to `redis-batch-app`.

Now we'll modify `add` function as follows:

```
function add ({author, quote}, cb) {
  const key = `Quotes: ${uuid()}`
  client
    .multi()
    .hmset(key, {author, quote})
    .sadd(`Author: ${params.author}`, key)
    .exec((err, replies) => {
      if (err) return cb(err)
      if (replies[0] === "OK") console.log('Added...\n')
      cb()
    })
}
```

We also need to alter the first `if` statement to account for the now asynchronous nature of the `add` function:

```
if (params.author && params.quote) {
  add(params, (err) => {
    if (err) throw err
    list((err) => {
      if (err) console.error(err)
      client.quit()
    })
  })
  return
}
```

The call to `client.multi` essentially puts the `redis` client into batch mode, queueing each subsequent command. When `exec` is called the list of commands is to sent to Redis and executed all in one go.

Using `hiredis`

By default, the redis module uses a pure JavaScript parser. However, the Redis project provides `hiredis` a module with Native C bindings to the official Redis client, Hiredis.

We may find performance gains (althogh mileage may vary), by using a parser written in C.

The `redis` module will avail of `hiredis` if it's installed, so to enable a potentially faster Redis client we can simply install the `hiredis` Node module:

```
$ npm install --save hiredis
```

Authenticating

We can set the authentication for Redis with the `redis.conf` file, found in the directory we installed Redis to.

To set a password in `redis.conf`, we simply uncomment the `require pass` section, supplying the desired password (for the sake of a concrete example let's choose the password "ourpassword").

Then, we make sure that our Redis server points to the configuration file.

If we are running it from the `src` directory, we would then start our redis server with the following command:

```
$ ./redis-server ../redis.conf
```

As an alternative if we want to quickly set a temporary password, we can use the following:

```
$ echo "requirepass ourpassword" | ./redis-server -
```

We can also set a password from within Node with the `CONFIG SET` Redis

command:

```
client.config('SET', 'requirepass', 'ourpassword')
```

To authenticate a Redis server within Node, we use the `redis` module's `client.auth` method before any other calls:

```
client.auth('ourpassword')
```

The password has to be sent before any other commands.

The `redis` module seamlessly handles re-authentication, we don't need to call client.auth again at failure points, this is taken care of internally.

See also

- TBD

Embedded Persistance with LevelDB

LevelDB is an embedded database developed at Google, and inspired by elements of Google's proprietary BigTable database. It's a log-structured key-value store purposed for fast read/write access of large data sets.

LevelDB has no command line or server interface, it's intended for use directly as a library. One of the advantages of an embedded database is we eliminate peer dependencies - we don't have to assume that a database is available at a certain host and port, we simply require a module and use the database directly.

In this recipe we're going to implement a quotes application on top of LevelDB.

Getting Ready

There's no external database to install, all we need to do is create a folder, with an `index.js`, initialize it as a package and install some dependencies:

```
$ mkdir level-app
$ cd level-app
$ touch index.js
```

```
$ npm init -y
$ npm install --save level xxhash end-of-stream-through2
```

How to do it

Let's start by loading our dependencies:

```
const {hash} = require('xxhash')
const through = require('through2')
const eos = require('end-of-stream')
const level = require('level')
```

The `xxhash` module is an implementaion of a very fast hashing algorithm which we'll be using in part to generate keys. The `through2` and `end-of-stream` modules are stream utility modules, and appear throughout this book.

The `level` module is a combination of LevelDOWN which provides native C++ bindings to the LevelDB embedded library, and LevelUP which provides a cohesive API layer.

Let's instantiate a LevelDB database:

```
const db = level('./data')
```

On first run this will create a `data` folder in our `level-app` directory.

We want to be able to add and list quotes, and we'll control our quotes application via the command line.

So to add a quote our command will be

```
$ node index.js "<Author>" "<Quote>"
```

To list quotes by a certain author, the command will be

```
$ node index.js "<Author>"
```

So let's implement the command line interface portion:

```
  const params = {
    author: process.argv[2],
    quote: process.argv[3]
  }

  if (params.author && params.quote) {
    add(params, (err) => {
      if (err) console.error(err)
      list(params.author)
    })
    return
  }

  if (params.author) {
    list(params.author)
    return
  }
```

In the previous snippet we reference to functions which are as yet unwritten, the `add` and `list` functions.

Let's write the `add` function:

```
  function add({quote, author}, cb) {
    const key = author + hash(Buffer.from(quote), 0xDAF1DC)
    db.put(key, quote, cb)
  }
```

Finally we'll implement the `list` function:

```
  function list (author) {
    if (!author) db.close()
    const quotes = db.createValueStream({
      gte: author,
      lt: String.fromCharCode(author.charCodeAt(0) + 1)
    })
    const format = through((quote, enc, cb) => {
      cb(null, `${author} ${quote}`)
    })
    quotes.pipe(format).pipe(process.stdout)
    eos(format, () => {
      db.close()
      console.log()
    })
  }
```

Now we should be able to test like so:

```
$ node index.js "Shaggy" "Like...no way man\!"
```

How it works

When we call `level` and pass it a path, a folder is created that holds all the data for our database.

Getting parameters from `process.argv` is fairly common fare, we load these values into an object containing `author` and `quote` properties.

If both `params.author` and `params.quote` are non-falsey we pass the `params` object to the `add` function which uses argument destructuring to assign the `author` and `quote` properties to function scoped variables of the same name within the `add` function.

We create a unique key for our quote by suffixing the author name with a hash of the quote, then we call `db.put` passing our `key` as the key, and the user supplied `quote` as the value.

The callback is called upon insertion, if there was an error we're sure to log it out, and then we go on to call the `list` function with the `params.author` value as the only input (the `list` function is also called straight away if the user does not supply a quote).

A fundamental principle of LevelDB is lexicographic sorting (JavaScript Arrays `sort` function is lexicographic which is why `[11, 100, 1].sort()` is `[ 1, 100, 11 ]`)).

If we cared about preserving insertion order in the `add` function we may have instead appended a persisted lexicographic counter, using the lexicographic-integer module.

Our `list` function calls `db.createValueStream` with an options object containing `gte` and `lt` properties, assigning the resulting stream to `quotes` .

The `gte` property instructs LevelDB to output all values whose keys are lexicographically greater than or equal to `author` (as ultimately provided by the user via the command line interface).

The `lt` property is set to the character which is the next code point up from the first character in the authors name. For instance if the author was "Adam Smith", the `lt` property would be the letter "B" (`String.fromCharCode('Adam Smith'.charCodeAt(0) + 1)` returns `'B'`). So here we're telling LevelDB to give us every key whose lexicographical value is less than (but not including) the next code point up.

In essence, this means our `quotes` stream will output all quotes by a given author. Beneath the `quotes` stream we create a `format` stream with the `through2` module, which essentially takes each quote and prefixes it with the `author` . Then we pipe from the `quotes` stream through the `format` stream to `process.stdout` .

The eager incremental processing afforded by LevelDB allows us to apply Node's stream paradigm over the top of LevelDB. Using streams means we can begin receiving, processing and sending results immediately, no matter how many results there are.

For the sake of aesthethetics its nice to have a newline at the end of all the output, to achieve this we use the `end-of-stream` module (assigned to `eos`) and pass the `format` stream to it. When the `format` stream is done we explicitly close the database and log a blank line. We cannot use `end-of-stream` on `process.stdout` because it has the unique quality of being an uncloseable stream.

There's more

Let's take a look at swapping out storage mechanisms.

Alternative Storage Adapters

The `levelup` module was separated from the `leveldown` module, so that various storage backends could be swapped in.

For instance, let's take our `level-app` folder from the main recipe, and copy it to a new folder, naming it `level-sql-app` :

```
$ cp -fr level-app level-sql-app
```

Now we'll remove the old `data` folder, unisntall `level` and install `levelup` , `sqlite` and `sqldown` :

```
$ cd level-sql-app
$ rm -fr data
$ npm uninst --save level
$ npm i --save levelup sqlite sqldown
```

Now we simply change the following lines from our main recipe:

```
const level = require('level')

const db = level('./data')
```

To the following:

```
const levelup = require('levelup')
const sqldown = require('sqldown')
const db = levelup('./data', {db: sqldown})
```

Our application will run in exactly the same way, except now we're storing to SQLite instead of LevelDB.

The `sqldown` module also supports MySql and Postgres, and there's a myriad of other alternative adapters (including backends that can use browser storage mechanisms) listed at https://github.com/Level/levelup/wiki/Modules#storage-back-ends.

See also

- TBD