R.E.S.T. API's (continued)

Building a books REST API

- GET
- POST
- DELETE
- PUT

GET Route

```
var express = require('express');
var app = express();
var books = require('./books');

// use the express.statis middle-ware to serve up static files on '/'
app.use(express.static('public'));

app.get('/books', function (request, response) {
    response.status(200).json(books.getBooks());
});

app.listen(3000, function () {
    console.log('listening on port 3000');
});
```

GET Route

```
var express = require('express');
var app = express();
var books = require('./books');

// use the express.statis middle-ware to serve up static files on '/'
app.use(express.static('public'));

app.get('/books', function (request, response) {
    response.send(books.getBooks());
});

app.get('/books/:id', function (request, response) {
    var id = request.params.id;

    var book = books.getBook(id);

    // check that you got a book back before sending it
    if (book) {
        response.send(book);
    // send a 404 status if you don't find it
    } else {
        response.status(404).json('Book not found');
    }
});
```

To create new books, we'll need to issue a POST request to the '/books' route, sending title, author, and genre.

Before we can read data from the client, we need to install the body-parser middle-ware which is not shipped with express.

```
// body-parser is a dependency and should be saved to package.json
npm install --save body-parser
```

The body-parser middle-ware will read the POSTED data from the request and create a new 'body' object containing the parsed data on the request object. It will contain key/value pairs where the value can be a string, array, or json data. Inside the route, form data is available on this request.body object.

We want to use the urlencoded method on body-parser to return a middle-ware function.

```
var express = requre('express');
var app = express();
var books = require('./books');
var bodyparser = require('body-parser');
var idManager = require('./idManager');

app.use(bodyparser.urlencoded({ extended: true });
app.use(bodyparser.json());

// previous code omitted for space
app.post('/books', function (request, response) {
    var newBook = request.body;

    if (newBook) {
        // modify book object here if desired
        // newBook._id = idManager.getId();
        // newBook.read = false;
        books.add(newBook);
        response.status(201).send(newBook);
    } else {
        response.status(400).send('problem adding book');
    }
});

app.listen(3000, function () {
    console.log('listening on port 3000');
});
```

Server should respond back with a **201** Created response code on successful creation of the new book record and return the new book as the response body, with any additional properties that were added during the adding process.

Implementing the DELETE route

Similar to the parameterized GET route, but with a different implementation. After deleting the data, the route should return a **200** response code (if returning OK) or **204** (if returning empty body.) Use the sendStatus function:

```
var express = require('express');
var app = express();
var books = require('./books');

// previous code omitted for space

app.delete('/books/:id', function (request, response) {
    var id = request.params.id;

    books.removeBook(bookId);
    response.sendStatus(200);
});

app.listen(3000, function () {
    console.log('Listening on port 3000');
});
```

Implementing a PUT Route

Similar to the parameterized GET and DELETE routes, but again with a different implementation. In this case, find the book first, update the properties on a temporary book object, then update the books list.

```
var express = require('express');
var app = express();
var books = require('./books');

// previous code omitted for space

app.put('/books/:id', function (request, response) {
    var book = books.getBook(request.params.id);
    var updatedBook;

    if (book && request.body) {
        updatedBook = books.updateBook(request.body);
        response.send(updatedBook);
    } else {
        reponse.status(404).json('Could not locate book for update');
    }
});

app.listen(3000, function () {
    console.log('Listening on port 3000');
});
```

The PATCH verb is very similar to PUT, but updates only a part of the record.

Our code is getting messy, and express provides us with a nice way to refactor it.

Currently we have some duplication in the appearance of our routes:

```
var express = require('express');
var app = express();
var books = require('./books');

app.get('/books', function (request, response) {
    // return all books
});

app.get('/books/:id', function (request, response) {
    // return book by id
});

app.post('/books', function (request, response) {
    // create a new book record
});

app.put('/books/:id', function (request, response) {
    // update an existing book record
});

app.delete('/books/:id', function(request, response) {
    // delete an existing book record
});
```

To avoid the duplication, we can use Route instances which allows the omission of the route from the individual handlers.

```
var express = require('express');
var app = express();
var books = require('./books');

var booksRoute = app.route('/books');

booksRoute.get(function (request, response) {
    // return all books
});

booksRoute.post(function (request, response) {
    // create a new book record
});
```

This syntax can be further shortened by getting rid of the intermediate variable, booksRoute, and using chaining to combine the routes.

```
var express = require('express');
var app = express();
var books = require('./books');

app.route('/books')
    .get(function (request, response) {
        // return all books
    })
    .post(function (request, response) {
        // create a new book record
    });
```

The same step can be taken for other, repetitive routes:

```
var express = require('express');
var app = express();
var books = require('./books');

app.route('/books')
    .get(function (request, response) {
        // return all books
    })
    .post(function (request, response) {
        // create a new book record
    });

app.route('/books/:id')
    .get(function (request, response) {
        // return book by id
    })
    .put(function (request, response) {
        // update an existing book
    })
    .delete(function (request, response) {
        // delete an existing book
    });
```

This is better, but we're still doing too much in our main application file. A single application can have lots of routes, and having too much code in one file is typically an indication you're doing it wrong.

Extracting routes to modules

Extracting routes to modules

```
var express = require('express');
var app = express();
var bookRoutes= require('./routes/bookRoutes');

app.use(express.static('public'));

// the bookRoutes is now a middle-ware callback
app.use('/books', bookRoutes);

app.listen(3000, function () {
    console.log('listening on port 3000');
});
```

Create a dedicated folder for routes

Inside bookRoutes.js we create an instance of an express Router, and export that router using module.exports

```
var express = require('express');

// the Router method returns an instance which can be mounted as middle-ware
var router = express.Router();


module.exports = router;
```

We also need to move the access to the books module and related book logic into this module as well.

```
var express = require('express');

// the Router method returns an instance which can be mounted as middle-ware
var router = express.Router();

// the path we mount the router on is relative to where it was mounted in app.
router.route('/')
    .get(function (request, response) {
        // get books
    })
    .post(function (request, response) {
        // create a new book record
    });

router.route('/:id')
    .get(function (request, response) {
        // get a book by id
    })
    .put(function (request, response) {
        // update an existing book
    })
    .delete(function (request, response) {
        // delete an existing book
    });

module.exports = router;
```