4. Building RESTful APIs using express

RESTful services

REST is short for REpresentational State Transfer. RESTful Web Services are basically REST Architecture based Web Services. In REST Architecture everything is a resource. RESTful web services are lightweight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications.

REST is web standards based architecture and uses HTTP Protocol for data communication. It revolves around a resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and presents the resources. Here each resource is identified by URIs/ global IDs. REST uses various representations to represent a resource like text, JSON and XML. Nowadays JSON is the most popular format being used in web services.

Following well known HTTP methods are commonly used in REST based architectures:

- GET: provides a read only access to a resource.
- POST: used to create a new resource.
- DELETE: used to remove a resource.
- PUT: used to update an existing resource or create a new resource.
- OPTIONS: used to get the supported operations on a resource

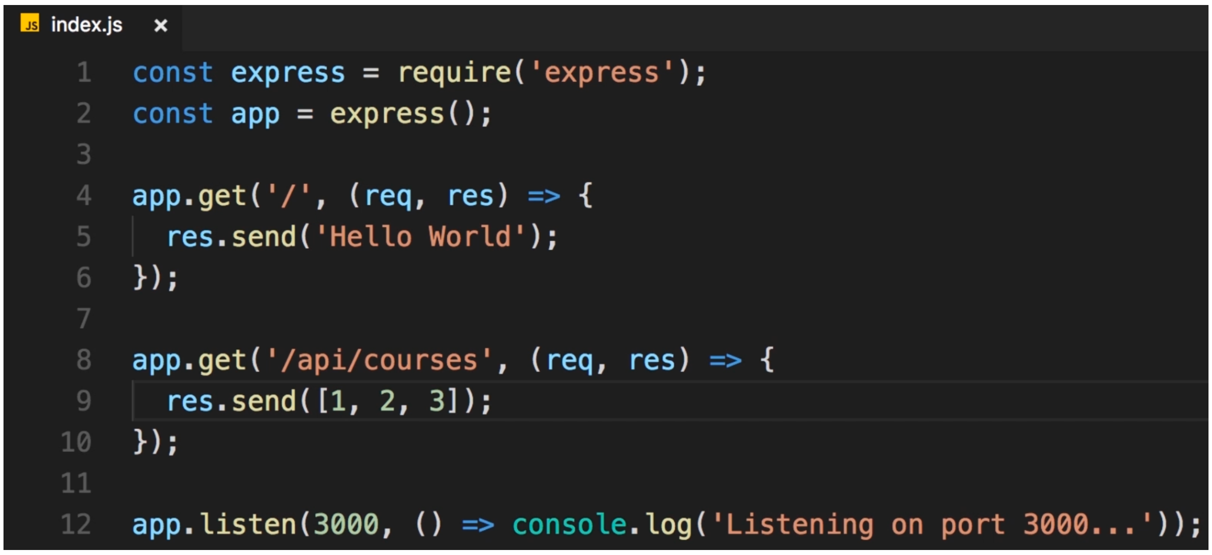
Introducing express

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It is a layer built on the top of Node.js that helps manage servers and routes. To install express simply run `npm i express`.

Then you can add it to a module:

```
const express = require('express');
const app = express();
```

An example of a simple express app:



index.js

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello World');
});

app.get('/api/courses', (req, res) => {
  res.send([1, 2, 3]);
});

app.listen(3000, () => console.log('Listening on port 3000...'));
```

Routing

Routing refers to how an application's endpoints (URIs) respond to client requests. You define routing using methods of the Express `app` object that correspond to HTTP methods; for example, `app.get()` to handle GET requests and `app.post()` to handle POST requests. For a full list, see `app.METHOD`. You can also use `app.all()` to handle all HTTP methods and `app.use()` to specify middleware as the callback function.

These routing methods specify a callback function (sometimes called "handler functions") called when the application receives a request to the specified route (endpoint) and HTTP method. In other words, the application "listens" for requests that match the specified routes and methods, and when it detects a match, it calls the specified callback function.

In fact, the routing methods can have more than one callback function as arguments. With multiple callback functions, it is important to provide *next* as an argument to the callback function and then call `next()` within the body of the function to hand off control to the next callback. The following code is an example of a very basic route.

```
const express = require('express')
const app = express()

// respond with "hello world" when a GET request is made to the
homepage
app.get('/', (req, res) => {
  res.send('hello world')
})
```

Nodemon

Nodemon (node-monitor) is a command-line tool that helps with the speedy development of Node.js applications. It monitors your project directory and automatically restarts your Node application when it detects any changes.

This means that you do not have to stop and restart your applications in order for your changes to take effect. You can simply write code, and test your application a few seconds later.

To install nodemon you can run

```
sudo npm i -g nodemon
```

Nodemon serves as a replacement for Node, and does not require any code changes within your application. Once it has installed, you can start your application with auto-restart, using the following command:

```
nodemon index.js
```

Environment variables

With Node’s environment variables, you can configure your applications outside of our codebase. Environment variables provide information about the environment in which the process is running. We use Node environment variables to handle sensitive data like passwords, which we shouldn’t hard code, or configuration details that might change between runs, like what port a server should listen on. In Node.js these special variables are accessible on the global `process.env` object.

Route parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the `req.params` object, with the name of the route parameter specified in the path as their respective keys.

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```

To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

```
app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params)
})
```

Route handlers

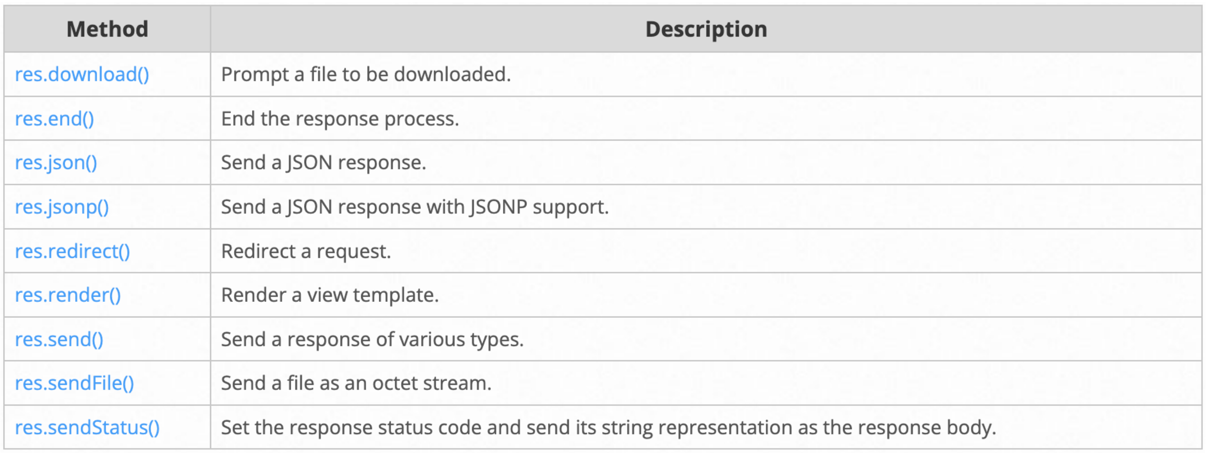
You can provide multiple callback functions that behave like middleware to handle a request. The only exception is that these callbacks might invoke `next('route')` to bypass the remaining route callbacks. You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route.

Route handlers can be in the form of a function, an array of functions, or combinations of both, as shown in the following example.

```
const cb0 = function (req, res, next) {
  console.log('CB0')
  next()
}

const cb1 = function (req, res, next) {
  console.log('CB1')
  next()
}

const cb2 = function (req, res) {
  res.send('Hello from C!')
}

app.get('/example/c', [cb0, cb1, cb2])
```

Response methods

The methods on the response object (`res`) in the following table can send a response to the client, and terminate the request-response cycle. If none of these methods are called from a route handler, the client request will be left hanging.



Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

HTTP response status codes

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

- Informational responses (100 – 199)
- Successful responses (200 – 299)
- Redirection messages (300 – 399)
- Client error responses (400 – 499)
- Server error responses (500 – 599)

The status codes listed below are defined by RFC 9110.

100 Continue
This interim response indicates that the client should continue the request or ignore the response if the request is already finished.

101 Switching Protocols
This code is sent in response to an Upgrade request header from the client and indicates the protocol the server is switching to.

200 OK
The request succeeded. The result meaning of "success" depends on the HTTP method:

- GET: The resource has been fetched and transmitted in the message body.
- HEAD: The representation headers are included in the response without any message body.
- PUT or POST: The resource describing the result of the action is transmitted in the message body.
- TRACE: The message body contains the request message as received by the server.

201 Created
The request succeeded, and a new resource was created as a result. This is typically the response sent after POST requests, or some PUT requests.

202 Accepted
The request has been received but not yet acted upon. It is noncommittal, since there is no way in HTTP to later send an asynchronous response indicating the outcome of the request. It is intended for cases where another process or server handles the request, or for batch processing.

203 Non-Authoritative Information
This response code means the returned metadata is not exactly the same as is available from the origin server, but is collected from a local or a third-party copy. This is mostly used for mirrors or backups of another resource. Except for that specific case, the 200 OK response is preferred to this status.

204 No Content
There is no content to send for this request, but the headers may be useful. The user agent may update its cached headers for this resource with the new ones.

300 Multiple Choices
The request has more than one possible response. The user agent or user should choose one of them. (There is no standardized way of choosing one of the responses, but HTML links to the possibilities are recommended so the user can pick.)

301 Moved Permanently
The URL of the requested resource has been changed permanently. The new URL is given in the response.

302 Found
This response code means that the URI of requested resource has been changed temporarily. Further changes in the URI might be made in the future. Therefore, this same URI should be used by the client in future requests.

303 See Other
The server sent this response to direct the client to get the requested resource at another URI with a GET request.

400 Bad Request
The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

401 Unauthorized
Although the HTTP standard specifies "unauthorized", semantically this response means "unauthenticated". That is, the client must authenticate itself to get the requested response.

402 Payment Required Experimental
This response code is reserved for future use. The initial aim for creating this code was using it for digital payment systems, however this status code is used very rarely and no standard convention exists.

403 Forbidden
The client does not have access rights to the content; that is, it is unauthorized, so the server is refusing to give the requested resource. Unlike 401 Unauthorized, the client's identity is known to the server.

404 Not Found
The server cannot find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid but the resource itself does not exist. Servers may also send this response instead of 403 Forbidden to hide the existence of a resource from an unauthorized client. This response code is probably the most well known due to its frequent occurrence on the web.

407 Proxy Authentication Required
This is similar to 401 Unauthorized but authentication is needed to be done by a proxy.

408 Request Timeout
This response is sent on an idle connection by some servers, even without any previous request by the client. It means that the server would like to shut down this unused connection. This response is used much more since some browsers, like Chrome, Firefox 27+, or IE9, use HTTP pre-connection mechanisms to speed up surfing. Also note that some servers merely shut down the connection without sending this message.

414 URI Too Long
The URI requested by the client is longer than the server is willing to interpret.

418 I'm a teapot
The server refuses the attempt to brew coffee with a teapot.

500 Internal Server Error
The server has encountered a situation it does not know how to handle.

501 Not Implemented
The request method is not supported by the server and cannot be handled. The only methods that servers are required to support (and therefore that must not return this code) are GET and HEAD.

502 Bad Gateway
This error response means that the server, while working as a gateway to get a response needed to handle the request, got an invalid response.

503 Service Unavailable
The server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded. Note that together with this response, a user-friendly page explaining the problem should be sent. This response should be used for temporary conditions and the Retry-After HTTP header should, if possible, contain the estimated time before the recovery of the service. The webmaster must also take care about the caching-related headers that are sent along with this response, as these temporary condition responses should usually not be cached.

504 Gateway Timeout
This error response is given when the server is acting as a gateway and cannot get a response in time.

505 HTTP Version Not Supported
The HTTP version used in the request is not supported by the server.