**Chapter 4**

**Restful Web Services with Express**

**Express**

In the previous chapter, we looked at the core of HTTP functionality provided by Node.js and saw how we can use the HTTP, Path and File module in Node JS to host our mini server.

In this chapter, we will be looking at how we can use another more powerful framework, Express to create our server application.

**Basics of Express**

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. ExpressJS is a prebuilt NodeJS framework that can help you in creating server-side web applications faster and smarter. Simplicity, minimalism, flexibility, scalability are some of its key features.

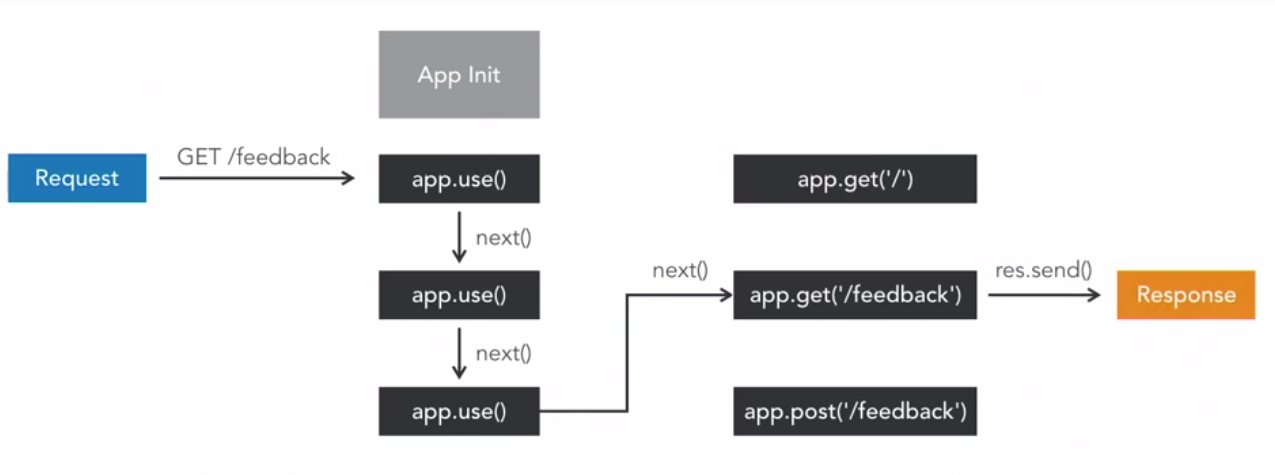
Express also supports various middleware modules which you can easily apply and use to perform additional tasks on request and response.

When the web server receives a request for data, Express provides a request object with information about the user and the data they sent in the request. Express also gives you access to a response object that you can modify before the web server responds to the user. These objects are normally shortened and named as req, res for the request and response object.

Middleware functions allow you to modify the req and res objects with relevant information. For instance, after a user has logged in, you could fetch their user details from a database, and then store those details in res.user.

In short, middleware gives you access to req and res in the apps request-> response cycle.

(Good explanation of what is middleware: <https://medium.com/@jamischarles/what-is-middleware-a-simple-explanation-bb22d6b41d01>)



Express is available on npm as express (npm install express). When you call require('express'), you get a function that you can call to create an express app.

**Serving static web pages**

Let us create a new application to serve static web pages, similar to the exercise we did in the previous chapter.

Copy first\_node\_server folder and its files to a new folder first\_express\_server. Delete the package.json and server.js files in the first\_express\_server folder.

Startup the terminal in visual studio code and type the following to initialize a package.json file in the first\_express\_server folder:

|  |
| --- |
| npm init |

Accept all default suggestions.

Next run:

|  |
| --- |
| npm install express --save |

You will notice the terminal displaying messages to show it downloading and installing the express library. The --save option parameter will save the dependency library for this application into the package.json file as shown below:

|  |
| --- |
| {  "name": "first\_express\_server",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "author": "",  "license": "ISC",  "dependencies": {  "express": "^4.16.4"  }  } |

The external libraries that you install through npm install will be saved in the node\_modules folder.

The package.json allows you to distribute your source code and files to your server machine or other machines without the need to copy the node\_modules folder as the package.json file, allows you to re-download the extra libraries when you type “npm install” in the new machine.

Create a new file server.js in the first\_express\_server directory:

|  |
| --- |
| var express = require('express');  var serveStatic = require('serve-static');  var app = express();  var port=3000;  var hostname="localhost";  app.use(serveStatic(\_\_dirname + '/public')); //apply middleware with app.use  app.listen(port, hostname, () => {  console.log(`Server started and accessible via http://${hostname}:${port}/`);  }); |

Finally run node server.js to test the application.

**Express Request object**

The Express request object derives from the Node.js request object we saw in the previous chapter. From the request object, we can retrieve incoming data from the client for processing. Example of some of the data we can retrieve from the client includes the url, method, cookie data and the query parameters sent from the client.

To test out some of the data we can retrieve from the client, modify server.js to include the below code just before we apply the static resource middleware.

|  |
| --- |
| …  app.use(**function(req, res, next)** {//create our custom middleware  console.log(req.url);  console.log(req.method)  console.log(req.path);  console.log(req.query.id);  next();  });  …. |

Try testing out server.js in the browser with <http://localhost:3000>, <http://localhost:3000/index.html?id=5> and observe the messages in the console.

**Express Response object**

The Express response object derives from the Node.js response object we saw in the previous chapter. The response object constitutes the data we return back to the client. We can use the response object to set the status, content-type and even redirect the user to another webpage.

Let us modify server.js to experiment what we can do with the response object.

|  |
| --- |
| …  app.use(function(req, res, next) {  console.log(req.url);  console.log(req.method)  console.log(req.path);  console.log(req.query.id);  res.status(200);  res.type(".html");  res.end("<html><body>Using response object!!</body></html>");  //res.redirect("https://www.sp.edu.sg");//comment out if we just want to redirect  //next();//since we are setting the whole response data, do not pass to the next middleware  });  … |

Rerun server.js and observe the effects of the code.

**What Are RESTful Web Services?**

REST is a term invented by Roy Fielding in his doctoral dissertation “Architectural Styles and the Design of Network-based Software Architecture” in 2000. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers (URIs)**, typically web url links. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

* **Resource identification through URI**: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery.
* **Uniform interface**: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE.
* **Self-descriptive messages**: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, but JSON is the most popular one.
* **Stateless interactions**: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

**HTTP Methods**

|  |  |  |
| --- | --- | --- |
| **Method** | **Operation performed on server** | **Quality** |
| GET | Read a resource. | Safe |
| PUT | Insert a new resource or update if the resource already exists. | Idempotent |
| POST | Insert a new resource. Also can be used to update an existing resource. | N/A |
| DELETE | Delete a resource. | Idempotent |

A Safe operation is an operation that does not have any effect on the original value of the resource. A read operation only fetches data, so it is safe and does not modify the original data in any way. An Idempotent operation is an operation that gives the same result no matter how many times you perform it. For example, if you delete a user with id 10, the result will still be the same regardless of how many times it is performed.

##### Difference between PUT and POST

The key difference between PUT and POST is that PUT is idempotent while POST is not. No matter how many times you send a PUT request, the results will be same. POST is not an idempotent method. Making a POST multiple times may result in multiple resources getting created on the server.

Another difference is that, with PUT, you must always specify the complete URI of the resource. This implies that the client should be able to construct the URI of a resource even if it does not yet exist on the server. This is possible when it is the client's job to choose a unique name or ID for the resource, just like creating a user on the server requires the client to choose a user ID. If a client is not able to guess the complete URI of the resource, then you have no option but to use POST.

|  |  |
| --- | --- |
| **Request** | **Operation** |
| PUT http://.../User/1 | Insert a new user with **Userid=1** if it does not already exist, or else update the existing resource |
|  |  |
| POST http://.../User/ | Insert a new person every time this request is made and generate a new **Userid**. |
| POST http://.../User/1 | Update the existing person where **Userid =1** |

From the above table, we can see that the PUT request will not modify or create more than one resource no matter how many times it is called. There is no difference between PUT and POST if the resource already exists, as both update the existing resource. The third request (POST http://.../User/) will create a new resource each time it is called.

For our exercises, we will be using GET for retrieval, POST for insertion, PUT for updating, and DELETE for deletion of resources.

**Express Application Routes**

Because of the importance of HTTP verbs when making RESTFUL web APIs, Express provides the HTTP verb + URL based routing support.

With Express, you can simply code and use app.VERB(path, callback) to register a middleware chain that is only called when the HTTP verb and PATH in the client request matches. Also you can call app.all to register a middleware that is called whenever the path matches (irrespective of the HTTP verb).

To illustrate the concept, let us assume we are supporting the 2 web service methods for GET and POST below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S/N | URL | HTTP Method | POST Body | Result |
| 1 | api/user | GET | empty | Retrieve data of all users |
| 2 | api/user | POST | JSON String of user details | Insert new user record |

As of now, we will just hardcode some response back to client and will not do the actual database data retrieval or insertion. We will include the database interaction in the next 2 chapters.

Let us create a new folder and project first\_webservice. Next run terminal in visual studio code and type the following commands:

|  |
| --- |
| npm init |

Accept all default suggestions.

Next run:

|  |
| --- |
| npm install express --save  npm install body-parser --save |

We will be using body-parser middleware library in order to read HTTP POST data. Body-parser is a piece of express middleware that reads a form's POST input data and stores it as a javascript object accessible through req.body.

With the libraries installed, let’s proceed to code out our RESTFul Web Service server app.

Let’s first create the route for api/user. Create a new file server.js and add in the following code:

|  |
| --- |
| var express = require('express');  var bodyParser = require('body-parser');  var port=8081;//use another port 8081 for this exercise  var hostname="localhost";  var app = express();  var urlencodedParser = bodyParser.urlencoded({ extended: false });  app.use(urlencodedParser);//attach body-parser middleware  app.use(bodyParser.json());//parse json data  //VERB+URL  app.get('/api/user', function (req, res) {  res.status(200);  res.type(".html");  res.end("Data of all users sent!");  });  app.listen(port, hostname, () => {  console.log(`Server started and accessible via http://${hostname}:${port}/`);  }); |

Run node server.js to test our application.

What url should we type in our browser to test the GET web service that we just created?

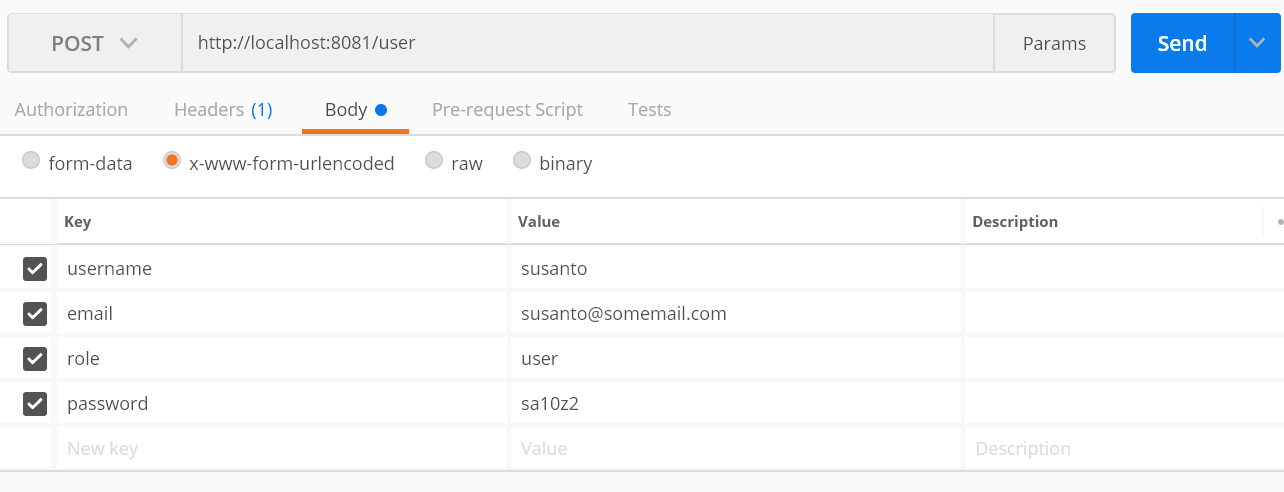
Activity:

Let’s code out the post method for url api/user. We will assume the client posts a request with 4 fields, username, email, role (Admin/user) and password. For now, attempt to extract out the input paramaters and return a string containing a message “Received new user data username <username>, email <email>, role <role> and password <password>”.

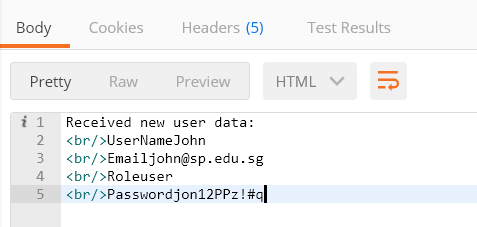
|  |
| --- |
| //assume user has a username, email, role(admin/user) and password  app.post('/api/user', function (req, res) {  ………..  }); |

**Testing the post webservice**

Startup POSTMAN and test the webservice by keying in the below fields and following the various selections:



After clicking on send, you should get the below result:



With that, we have seen how using express makes it easy to host webservices. In the next chapter, we will be combining database and web services to create the backend of the web application.