EXPRESS APPLICATION

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework –

- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically render HTML Pages based on passing arguments to templates.

Installing Express

Firstly, install the Express framework globally using NPM so that it can be used to create a web application using node terminal.

```
$ npm install express --save
```

- **body-parser** – This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- **cookie-parser** – Parse Cookie header and populate req.cookies with an object keyed by the cookie names.
- **multer** – This is a node.js middleware for handling multipart/form-data.

```
$ npm install body-parser --save
$ npm install cookie-parser --save
$ npm install multer --save
```

Hello world Example:

Following is a very basic Express app which starts a server and listens on port 8081 for connection. This app responds with Hello World! for requests to the homepage.

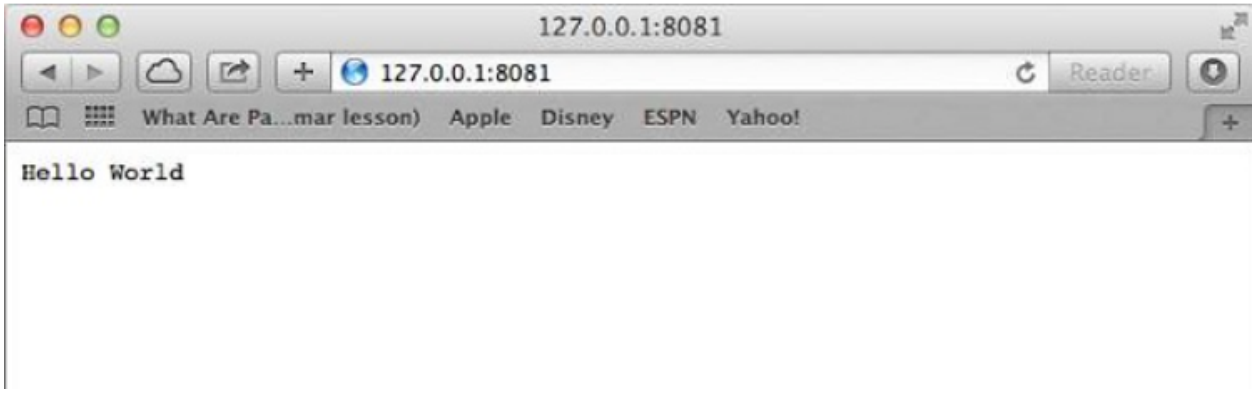
```
var express = require('express');
var app = express();

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

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Example app listening at http://%s:%s", host, port)
})
```

Save the above code in a file named server.js and run it with the following command.

$ node server.js

You will see the following output –

```
Example app listening at http://0.0.0.0:8081
Open http://127.0.0.1:8081/ in any browser to see the following result.
```



Request & Response

Express application uses a callback function whose parameters are request and response objects.

```
app.get('/', function (req, res) {
  // --
})
```

- Request Object – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- Response Object – The response object represents the HTTP response that an Express app sends when it gets an HTTP request.

Basic Routing

Following code in app.js binds two route handlers.

```
var routes = require('./routes/index');
var users = require('./routes/users');
...
app.use('/', routes);
app.use('/users', users);
```

1. routes - routes index.js, route handler handles all request made to home page via localhost:3000

2. users - users users.js, route handler handles all request made to /users via localhost:3000/users

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

Node.js – Enhancing First Application

In this article, we'll enhance the Express.js application created in the Express Application chapter to include the following functionalities:

1. Show a list of all users.

2. Show details of a particular user.

3. Add details of a new user.

Step 1: Create a JSON based database

Firstly, let's create a sample JSON based database of users.

```
{
   "user1" : {
      "name" : "mahesh",
   "password" : "password1",
   "profession" : "teacher"
   },
   "user2" : {
      "name" : "suresh",
   "password" : "password2",
   "profession" : "librarian"
   },
   "user3" : {
      "name" : "ramesh",
   "password" : "password3",
   "profession" : "clerk"
   }
}
```

Step 2: Create Users specific Jade Views

Create a user directory in following views.

1. index .jade – View to show a list of all users.

2. new .jade – View to display a form for adding a new user.

3. profile .jade – View to show details of a user.

```
h1 Users

p
   a(href="/users/new/") Create new user
ul
   - for (var username in users) {
   li
      a(href="/users/" + encodeURIComponent(username))= users[username].name
   - };
```

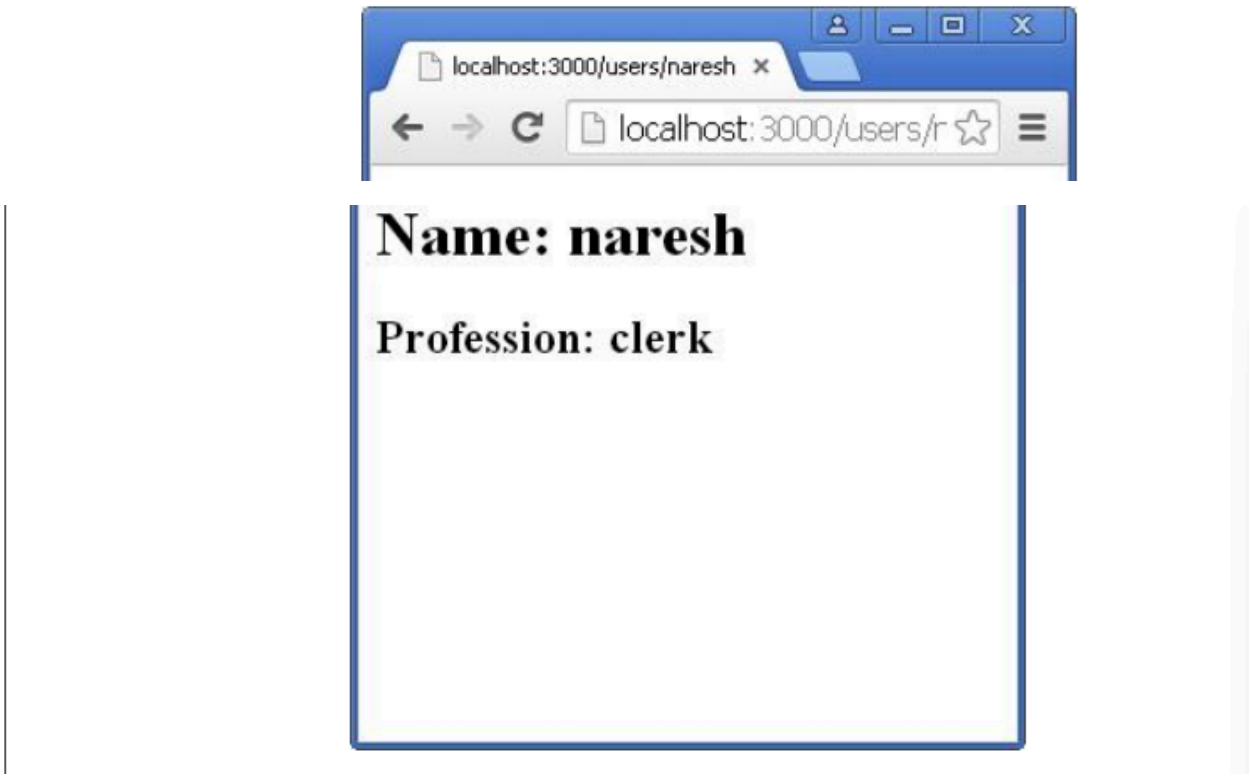
```
h1 New User

form(method="POST" action="/Users/addUser")
 P
  label(for="name") Name

  input#name(name="name")
 P
```

```
  label(for="password") Password

  input#name(name="password")
 P
  label(for="profession") Profession

  input#name(name="profession")
 P
  input(type="submit", value="Create")
```

Step 3: Update users route handler, users.js

```
var express = require('express');
var router = express.Router();

var users = require('../users.json');
/* GET users listing. */
router.get('/', function(req, res) {
   res.render('users/index', { title: 'Users',users:users });
});

/* Get form to add a new user*/
router.get('/new', function(req, res) {
   res.render('users/new', { title: 'New User'});
});

/* Get detail of a new user */
router.get('/:name', function(req, res, next) {
   var user = users[req.params.name]
   if(user){
      res.render('users/profile', { title: 'User Profile', user:user});
   }else{
      next();
   }
});

/* post the form to add new user */
router.post('/addUser', function(req, res, next) {
   if(users[req.body.name]){
      res.send('Conflict', 409);
   }else{
     users[req.body.name] = req.body;
  res.redirect('/users/');
   }
});

module.exports = router;
```

Click on newly created user to see the details.



You can check the server status also as following:

```
C:\Nodejs_WorkSpace\firstApplication>node app
Express server listening on port 3000
GET /users/ 200 809.161 ms - 201
GET /users/new/ 304 101.627 ms - -
GET /users/new/ 304 33.496 ms - -
POST /Users/addUser 302 56.206 ms - 70
GET /users/ 200 43.548 ms - 245
GET /users/naresh 200 12.313 ms - 47
```

Node.js - RESTful API

REST (Representational State Transfer) is a web architecture using HTTP to access and manage resources. Introduced by Roy Fielding (2000), it follows standard methods and identifies resources via URIs. A REST Server provides resources, and a REST Client interacts with them. Common formats include JSON (most popular), XML, and Text.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – This is used to provide a read only access to a resource.
- **PUT** – This is used to create a new resource.
- **DELETE** – This is used to remove a resource.
- **POST** – This is used to update a existing resource or create a new resource.

Creating RESTful for A Library

Consider we have a JSON based database of users having the following users in a file,

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },

   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
      "id": 2
   },

   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
}
```

Based on this information we are going to provide following RESTful APIs.

Sr.No.	URI	HTTP Method	POST body	Result
1	listUsers	GET	empty	Show list of all the users.
2	addUser	POST	JSON String	Add details of new user.
3	deleteUser	DELETE	JSON String	Delete an existing user.
4	:id	GET	empty	Show details of a user.

Add details of new user.

When a client send a POST request to /users/addUser with body containing the JSON String, server should send a response stating the status. Update users route handler, users.js

```
/*add a user*/
router.post('/addUser', function(req, res, next) {
    var body = '';
    req.on('data', function (data) {
        body += data;
    });
    req.on('end', function () {
    var json = JSON.parse(body);
```

```
        users["user"+json.id] = body;
     res.send({ Message: 'User Added'});
     });
});
```

Show details of new user.

When a client send a GET request to /users with an id, server should send a response containing detail of that user. Update users route handler, users.js

```
router.get('/:id', function(req, res, next) {
   var user = users["user" + req.params.id]
   if(user){
      res.send({ title: 'User Profile', user:user});
   }else{
      res.send({ Message: 'User not present'});
   }
});
```

```
var express = require('express');
var router = express.Router();

var users = require('../users.json');
/* GET users listing. */
router.get('/', function(req, res) {
   res.send({ title: 'Users',users:users });
});

router.get('/:id', function(req, res, next) {
   console.log(req.params)
   var user = users["user" + req.params.id]
   if(user){
      res.send({ title: 'User Profile', user:user});
   }else{
      res.send({ Message: 'User not present'});
   }
});

router.post('/addUser', function(req, res, next) {
    var body = '';
    req.on('data', function (data) {
        body += data;
    });
    req.on('end', function () {
    var json = JSON.parse(body);
       users["user"+json.id] = body;
    res.send({ Message: 'User Added'});
    });
});

module.exports = router;
```

Output

We are using Postman, a Chrome extension, to test our webservices.

Now run the app.js to see the result:

```
C:\Nodejs_WorkSpace\firstApplication>node app
```

Verify the Output. Server has started

```
Express server listening on port 3000
```

Node.js - Scaling Application

Node.js runs in single-thread mode but uses an event-driven approach for concurrency. It supports child processes to utilize multi-core CPUs for parallel processing.

Each child process has stdin, std out, and stderr streams, which can be shared with the parent process. The child-process module provides three main methods to create child processes.

- **exec** – child_process.exec method runs a command in a shell/console and buffers the output.
- **spawn** – child_process.spawn launches a new process with a given command.
- **fork** – The child_process.fork method is a special case of the spawn() to create child processes.

The exec() method

child_ process .exec method runs a command in a shell and buffers the output. It has the following signature –

child_ process .exec(command[, options], callback)

Parameters

Here is the description of the parameters used –

- **command** (String) The command to run, with space-separated arguments
- **options** (Object) may comprise one or more of the following options –
    - **cwd** (String) Current working directory of the child process
    - **env** (Object) Environment key-value pairs
    - **encoding** (String) (Default: 'utf8')
    - **shell** (String) Shell to execute the command with (Default: '/bin/sh' on UNIX, 'cmd.exe' on Windows, The shell should understand the -c switch on UNIX or /s /c on Windows. On Windows, command line parsing should be compatible with cmd.exe.)
    - **timeout** (Number) (Default: 0)
    - **maxBuffer** (Number) (Default: 200*1024)
    - **killSignal** (String) (Default: 'SIGTERM')
    - **uid** (Number) Sets the user identity of the process.
    - **gid** (Number) Sets the group identity of the process.
- **callback** The function gets three arguments **error**, **stdout,** and **stderr** which are called with the output when the process terminates.

Example

Let us create two JS files named support.js and master.js –

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var workerProcess = child_process.exec('node worker.js '+i,
      function (error, stdout, stderr) {
         if (error) {
            console.log(error.stack);
            console.log('Error code: '+error.code);
            console.log('Signal received: '+error.signal);
         }
         console.log('stdout: ' + stdout);
         console.log('stderr: ' + stderr);
      });

      workerProcess.on('exit', function (code) {
      console.log('Child process exited with exit code '+code);
   });
}
```

Verify the Output. Server has started

```
Child process exited with exit code 0
stdout: Child Process 1 executed.

stderr:
Child process exited with exit code 0
stdout: Child Process 0 executed.

stderr:
Child process exited with exit code 0
stdout: Child Process 2 executed.
```

The spawn() Method

child_process.spawn method launches a new process with a given command. It has the following signature –

```
child_process.spawn(command[, args][, options])
```

- **command** (String) The command to run
- **args** (Array) List of string arguments
- **options** (Object) may comprise one or more of the following options –
    - **cwd** (String) Current working directory of the child process.
    - **env** (Object) Environment key-value pairs.
    - **stdio** (Array) String Child's stdio configuration.
    - **customFds** (Array) Deprecated File descriptors for the child to use for stdio.
    - **detached** (Boolean) The child will be a process group leader.
    - **uid** (Number) Sets the user identity of the process.
    - **gid** (Number) Sets the group identity of the process.

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var workerProcess = child_process.spawn('node', ['worker.js', i]);

   workerProcess.stdout.on('data', function (data) {
      console.log('stdout: ' + data);
   });

   workerProcess.stderr.on('data', function (data) {
      console.log('stderr: ' + data);
   });

   workerProcess.on('close', function (code) {
      console.log('child process exited with code ' + code);
   });
}
```

Verify the Output. Server has started

```
stdout: Child Process 0 executed.

child process exited with code 0
stdout: Child Process 2 executed.

child process exited with code 0
stdout: Child Process 1 executed.

child process exited with code 0
```

The fork() Method

child_process.fork method is a special case of spawn() to create Node processes. It has the following signature –

- **modulePath** (String) The module to run in the child.
- **args** (Array) List of string arguments
- **options** (Object) may comprise one or more of the following options –
    - **cwd** (String) Current working directory of the child process.
    - **env** (Object) Environment key-value pairs.
    - **execPath** (String) Executable used to create the child process.
    - **execArgv** (Array) List of string arguments passed to the executable (Default: process.execArgv).
    - **silent** (Boolean) If true, stdin, stdout, and stderr of the child will be piped to the parent, otherwise they will be inherited from the parent, see the "pipe" and "inherit" options for spawn()'s stdio for more details (default is false).
    - **uid** (Number) Sets the user identity of the process.
    - **gid** (Number) Sets the group identity of the process.

Example

Create two js file named worker.js and master.js

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var worker_process = child_process.fork("worker.js", [i]);

   worker_process.on('close', function (code) {
      console.log('child process exited with code ' + code);
   });
}
```

Verify the Output. Server has started

```
Child Process 0 executed.
Child Process 1 executed.
Child Process 2 executed.
child process exited with code 0
child process exited with code 0
child process exited with code 0
```