Scaling Applications with Node.js

Forking process:

Because Node.js is single threaded, we've run into the need to scale our applications much sooner that we would if we were using another programming language. It's not a problem. This is actually a benefit of Node because we get to talk about scaling much earlier in the application lifecycle because Node applications are made to scale. Node.js is designed to clone your application and then run it using multiple instances simultaneously. This process is called forking.

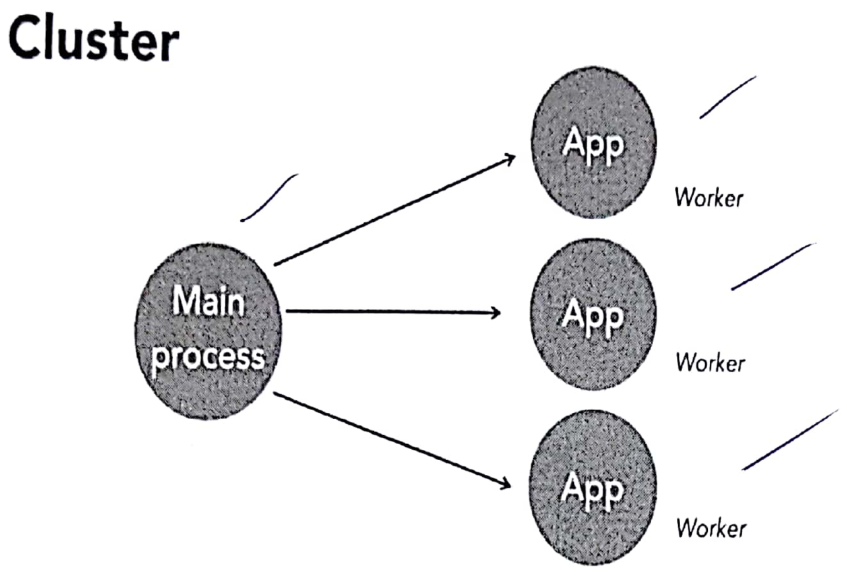
```
//server.js
const http = require("http");
const port = parseInt(process.argv[2] || '3000');
const options=[
    "Go for it",
    "Maybe sleep on it",
    "Do some more reasearch",
    "I don't know",
    "I wouldn't"
];
const server = http.createServer((req,res)=>{
    const randomIndex = Math.floor(Math.random() * options.length);
    const payload = JSON.stringify({
        port,
        processID:process.pid,
        advice:options[randomIndex]
    });
    res.writeHead(200,{'Content-type':"application/json"});
    res.end(payload);
});
server.listen(port);
console.log("advice server is running on port",port)
```

```
//index.js
const { fork } = require("child_process");

const processes = [
    fork("./server",['3001']),
    fork("./server",['3002']),
    fork("./server",['3003'])
]
console.log(`forked ${processes.length} process`);
//output
D:\devops\devopsportal\clone>node .
```

```
forked 3 process
advice server is running on port 3001
advice server is running on port 3002
advice server is running on port 3003
```

Using a cluster module:



Node.js, a single application instance only uses one processor because Node.js is single threaded. Forking your application into multiple instances is required to take full advantage of your hardware. A cluster is a group of node instances that all work together. A cluster is made up of worker processes, the four instances of our app, in a main process, which is the instance that spawns and controls the workers. Let's create a cluster to take advantage of every CPU that is available to us.

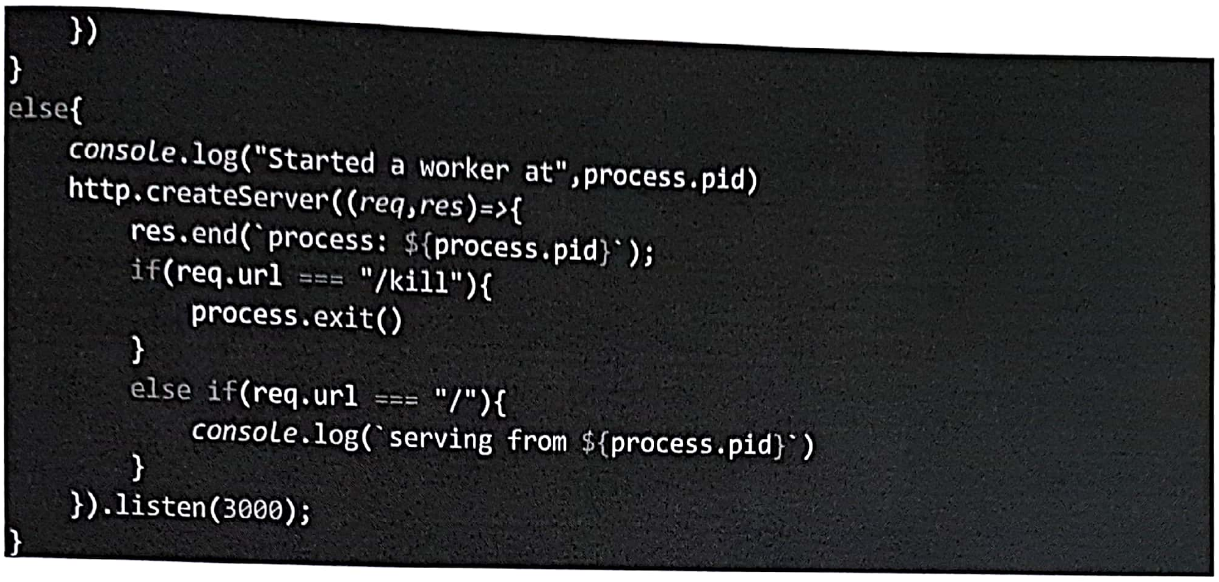
```
const cluster = require("cluster")
const cpus = require("os").cpus().length;
if(cluster.isMaster){
    console.log("I am in master process",process.pid);
    cluster.fork();
    cluster.fork();
    cluster.fork();
}
else{
    console.log("I am in child process",process.pid)
}
```

```
//index.js
const cluster = require("cluster");
const http = require("http");
const numCpus = require("os").cpus().length;
if(cluster.isMaster){
    console.log("I am in master process",process.pid);
    for(let i=0;i<numCpus;i++){
        cluster.fork();
    }
}
else{
    console.log("I am in child process",process.pid)
    http.createServer((req,res)=>{
        const message = `worker ${process.pid}....`
        console.log(message);
        res.end(message);
    }).listen(3000);
}
```

Architecting zero downtime:

One of the advantages of running multiple processes is that your app never has to go down, it can always be available to your users, this is called zero downtime. Sometimes our apps go down, this could be due to some mysterious and obscure bug, it could be due to high traffic or sometimes we just need to update the code and restart the process, in a cluster when a single instance fails, the traffic will use the remaining worker instances, and the main process can detect worker failures and automatically restart those workers. When you need to update your app you no longer need to tell your customers that the website will be down due to maintenance.

```
//index.js
const cluster = require("cluster");
const http = require("http");
const numCpus = require("os").cpus().length;
if(cluster.isMaster){
    console.log("I am in master process",process.pid);
    for(let i=0;i<numCpus;i++){
        cluster.fork();
    }
    cluster.on("exit",worker=>{
        console.log(`worker process ${process.pid} died`);
       // console.log(`only ${Object.keys(cluster.workers).length} are left`)
       console.log("starting new worker");
       cluster.fork();
```



```
    })
}
else{
    console.log("Started a worker at",process.pid)
    http.createServer((req,res)=>{
        res.end(`process: ${process.pid}`);
        if(req.url === "/kill"){
            process.exit()
        }
        else if(req.url === "/"){
            console.log(`serving from ${process.pid}`)
        }
    }).listen(3000);
}
```

Working with cluster with PM2:

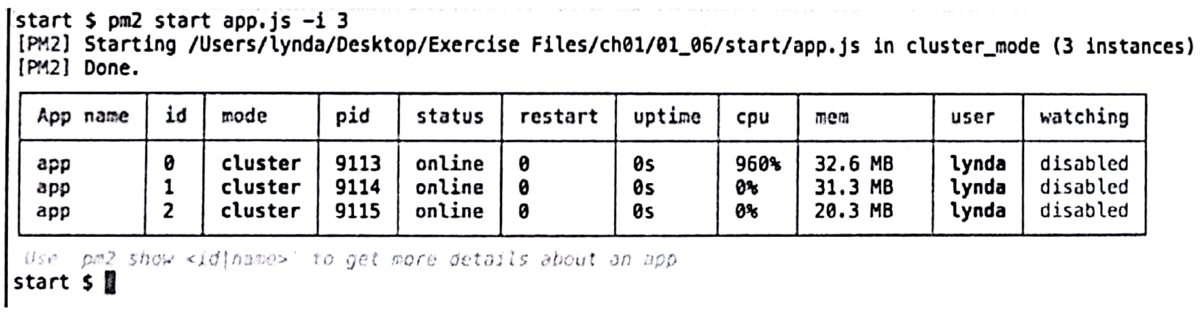
PM2 is Node.js process manager. It will allow you to manage zero down time clusters in production.

npm install pm2 –g

start pm2 with server.js file

pm2 start server.js –i 3

And what this will do is run three instances of the server. js and node. So we can see it's running right there.



```
start $ pm2 start app.js -i 3
[PM2] Starting /Users/lynda/Desktop/Exercise Files/ch01/01_06/start/app.js in cluster_mode (3 instances)
[PM2] Done.
┌──────────┬────┬─────────┬──────┬────────┬─────────┬────────┬──────┬─────────┬───────┬──────────┐
│ App name │ id │ mode    │ pid  │ status │ restart │ uptime │ cpu  │ mem     │ user  │ watching │
├──────────┼────┼─────────┼──────┼────────┼─────────┼────────┼──────┼─────────┼───────┼──────────┤
│ app      │ 0  │ cluster │ 9113 │ online │ 0       │ 0s     │ 960% │ 32.6 MB │ lynda │ disabled │
│ app      │ 1  │ cluster │ 9114 │ online │ 0       │ 0s     │ 0%   │ 31.3 MB │ lynda │ disabled │
│ app      │ 2  │ cluster │ 9115 │ online │ 0       │ 0s     │ 0%   │ 20.3 MB │ lynda │ disabled │
└──────────┴────┴─────────┴──────┴────────┴─────────┴────────┴──────┴─────────┴───────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app
start $ ▌
```

Pm2 list ->list all running ones

Pm2 stop server.js - > stops all running ones.

Pm2 delete server.js-> remove server from pm2

Pm2 monit -> monitor the logs

Pm2 start server.js –i -1

So when you use dash i negative one, PM2 will actually automatically select the number of instances that it should run for your current processor. So here you can see it's running seven instances of our application.

Loadtest:

Used to test the node load

npm install loadtest –g

once install , now test the application

loadtest –n 2000 http://localhost:3000

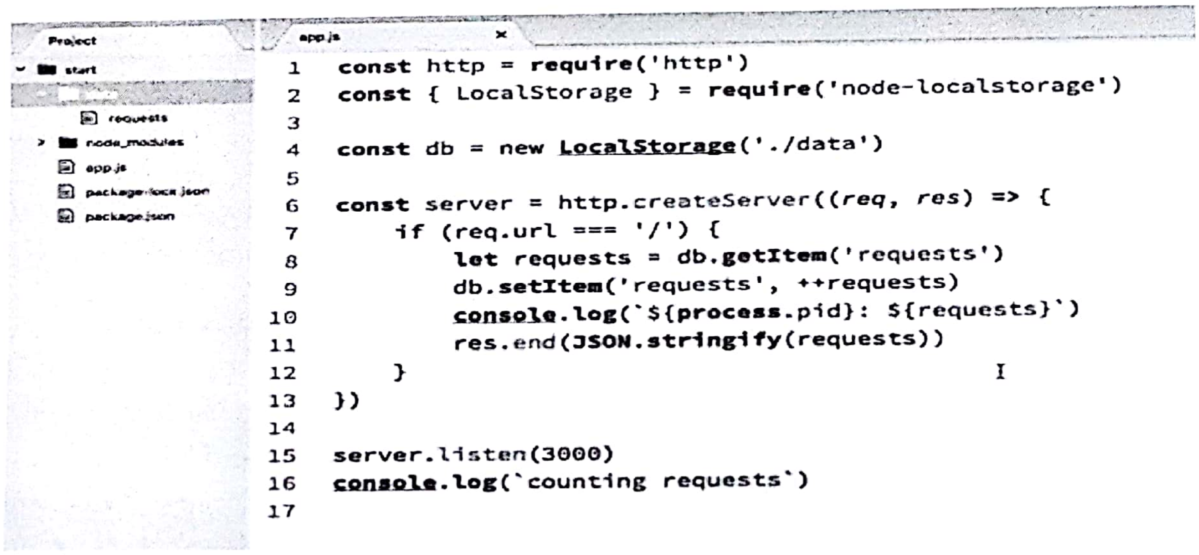
here 2000 means number of request

loadtest -n we'll go ahead and do 2000 http://localhost:3000 and running this will simulate 2000 requests to localhost 3000.

And, as this happens, all of these requests have been spanned out across our cluster. We can actually see that by running clear and PM2 listand we can notice that we're using a little bit more of the CPU and a little bit more memory for each of our processes to handle those 2000 requests. So, something else we might want to do is check the logs. PM2 logs will show you the logs for all of the processes. So you can see here that each number of process this is the list of the last fifteen logs that we can see.

If you change anything in server.js then pm2 reload server.js will restart all the clusters automatically.

Database scaling:



```
const http = require('http')
const { LocalStorage } = require('node-localstorage')

const db = new LocalStorage('./data')

const server = http.createServer((req, res) => {
    if (req.url === '/') {
        let requests = db.getItem('requests')
        db.setItem('requests', ++requests)
        console.log(`${process.pid}: ${requests}`)
        res.end(JSON.stringify(requests))
    }
})

server.listen(3000)
console.log(`counting requests`)
```