part3

STREAMS

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are three types of streams −

- **Readable** − Stream which is used for read operation.
- **Writable** − Stream which is used for write operation.
- **Duplex** − Stream which can be used for both read and write operation.

Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times. For example, some of the commonly used events are −

- **data** − This event is fired when there is data is available to read.
- **end** − This event is fired when there is no more data to read.
- **error** − This event is fired when there is any error receiving or writing data.
- **finish** − This event is fired when all the data has been flushed to underlying system.

Reading from a Stream

Step 1) Create a file called data.txt which has some text

Step 2) Write the relevant code which will make use of streams to read data from the file.

```
var fs = require("fs");
 var stream;
 stream = fs.createReadStream("D://data.txt");
stream.on("data", function(data) {
var chunk = data.toString();
 console.log(chunk);
 });
```

```
var fs = require("fs");
var datas = '';
var readerStream = fs.createReadStream('input.txt');
// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');
// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
   datas += chunk;
});
readerStream.on('end',function(){
   console.log(datas);
});
readerStream.on('error', function(err){
   console.log(err.stack);
});
console.log("Program Ended");
```

Node.js Writing to stream

```
var fs = require("fs");
var data = 'A Solution of all Technology';
var writerStream = fs.createWriteStream('output.txt’);

writerStream.write(data,'UTF8');

writerStream.end();
writerStream.on('finish', function() {
    console.log("Write completed.");
});
writerStream.on('error', function(err){
   console.log(err.stack);
});
console.log("Program Ended");
```

Node.js Piping Streams

Piping is a mechanism where output of one stream is used as input to another stream. There is no limit on piping operation.

```
var fs = require("fs");
// Create a readable stream
var readerStream = fs.createReadStream('input.txt');
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Pipe the read and write operations
// read input.txt and write data to output.txt
readerStream.pipe(writerStream);
console.log("Program Ended");
```

Node.js Chaining Streams

Chaining stream is a mechanism of creating a chain of multiple stream operations by connecting output of one stream to another stream. It is generally used with piping operation.

```
var fs = require("fs");
var zlib = require('zlib');
// Compress the file input.txt to input.txt.gz
fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('input.txt'));
  console.log("File Compressed.");
```

What is Routing?

Routing defines the way in which the client requests are handled by the application endpoints.

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

// Create a server object
http.createServer(function (req, res) {

   // http header
   res.writeHead(200, {'Content-Type': 'text/html'});

   var url = req.url;

   if(url ==='/about') {
      res.write(' Welcome to about us page');
      res.end();
   }
   else if(url ==='/contact') {
      res.write(' Welcome to contact us page');
      res.end();
   }
   else {
      res.write('Hello World!');
      res.end();
   }
}).listen(3000, function() {

   // The server object listens on port 3000
   console.log("server start at port 3000");
});
```

The package.json

The `package.json` file is kind of a manifest for your project.

It can do a lot of things, completely unrelated.

It's a central repository of configuration for tools, for example.

It's also where `npm` store the names and versions for all the installed packages.

The package.json file is the heart of Node.js system.

The metadata information in **package.json** file can be categorized into below categories:

1. **Identifying metadata properties:** It basically consist of the properties to identify the module/project such as the name of the project, current version of the module, license, author of the project, description about the project etc.
2. **Functional metadata properties:** As the name suggests, it consists of the functional values/properties of the project/module such as the entry/starting point of the module, dependencies in project, scripts being used, repository links of Node project etc.

nodemon

nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.

nodemon does not require *any* additional changes to your code or method of development.

nodemon is a replacement wrapper for node.

To use nodemon, replace the word node on the command line when executing your script.

Installation

```
npm install -g nodemon
```

Usage

```
nodemon [your node app]
```