Modules in NodeJS

Maharishi International University - Fairfield, Iowa

nextTick and Promise Queues

- All callbacks in the nextTick queue are executed before callbacks in the promise queue.
- However, if the current execution is in the Promise microtask queue, so it will not be interrupted by a later nextTick task that is scheduled afterward.
- If nextTick and Promise Queues are empty, then event loop (timers, IO, check, and close) queues are started to execute.

What will be the output?

```
process.nextTick(() => console.log("this is process.nextTick 1"));
process.nextTick(() => {
    console.log("this is process.nextTick 2");
    process.nextTick(() =>
        console.log("this is the inner next tick inside next tick")
    );
});
process.nextTick(() => console.log("this is process.nextTick 3"));

Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
Promise.resolve().then(() => {
    console.log("this is Promise.resolve 2");
    process.nextTick(() =>
        console.log("this is the inner next tick inside Promise then block")
    );
});
Promise.resolve().then(() => console.log("this is Promise.resolve 3"));
```

Execution order:

1. All sync codes are executed
2. nextTick & promise queue
3. timers phase
4. nextTick & promise queue after every callback in the timer queue.
5. I/O phase
6. nextTick & promise queue after every callback in the I/O queue.
7. check phase
8. nextTick & promise queue after every callback in the check queue.
9. close phase
10. nextTick & promise queue after every callback in the close queue.

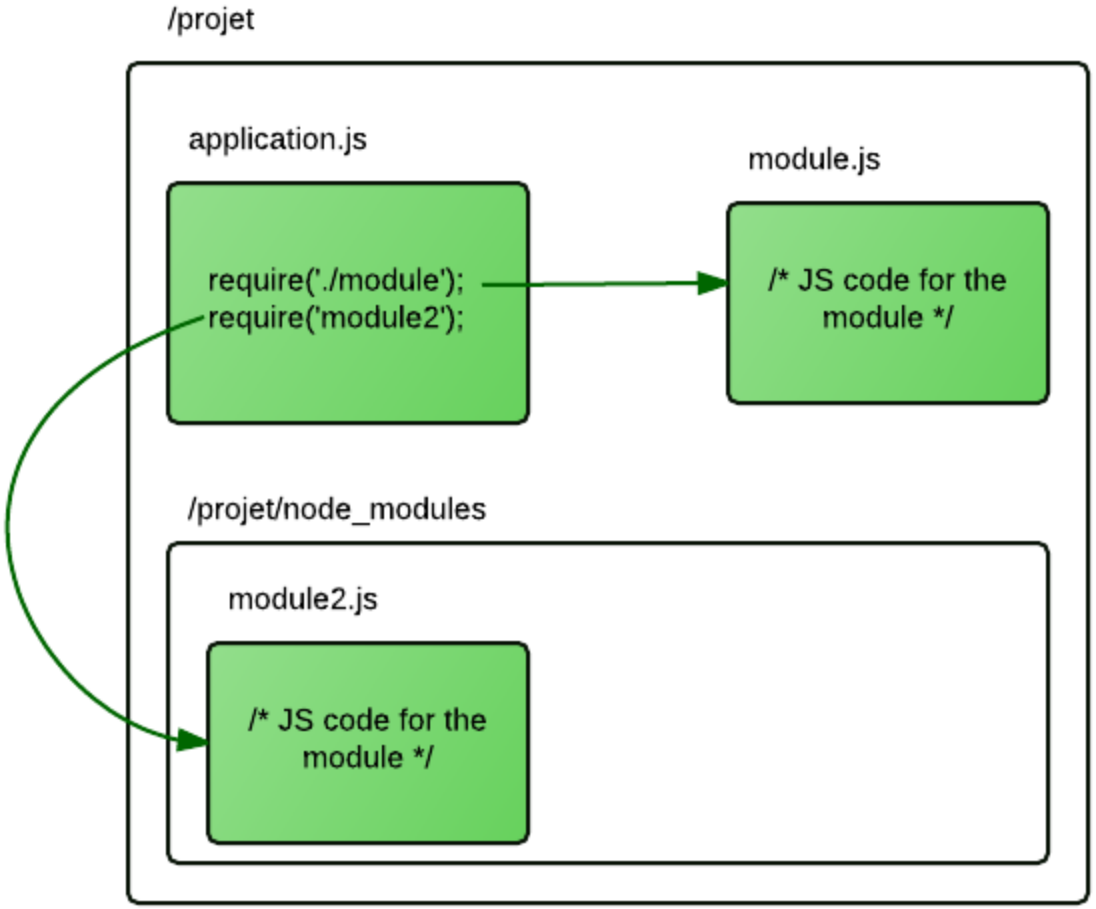
What will be the output?

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

process.nextTick(() => { console.log('nextTick 1') });
Promise.resolve().then(() => console.log('promise 1'));
setImmediate(() => {
    console.log('setImmediate 1')
    process.nextTick(() => console.log('nextTick fn'));
    Promise.resolve().then(() => console.log('promise fn'));
});
setTimeout(() => console.log('setTimeout 1'), 0);
fs.readFile('./files/input.txt', "utf-8", (err, data) => {
    if (err)
        console.log('there is an error. can not read from file');
    else {
        console.log(data);

    }
});
```

- In Node.js, there are **multiple ways to read from a file**, depending on whether you want **synchronous**, **asynchronous**, **streaming**, or **promise-based** behavior.
  - fs.readFileSync() — Synchronous
  - fs.readFile() — Async with Callback
  - fs.promises.readFile() — Async with Promise
  - fs.createReadStream() — Stream

Modules in NodeJS

- A module is an encapsulated and reusable chunk of code that has its own context.
- In Node.js, each file is treated as a separate module.
- There are two primary types of modules in Node.js:
  1. **CommonJS Modules** (used by default)
  2. **ES Modules** (the newer ECMAScript standard)



CommonJS Modules:

- Key Features:
  - Synchronous
  - Uses `require()` to import modules.
  - Uses `module.exports` or `exports` to export functionality.
  - Browsers don't support
- The module system has been the default in Node.js.
  - At the time node.js was created, there was no built-in module system in JavaScript.
  - So, Node.js defaulted to Common JS as its module system.

ES Modules:

- Key Features:
  - Asynchronous
  - Uses `import`/`export` keywords.
  - Supports top-level await.
  - Requires explicit file extensions.
  - Browsers support
- It is the modern JavaScript module system (ES6). They are now natively supported in Node.js starting with version 12+ (2022).

ES Module `import` Syntax

- **Static Imports**
  - Static imports are used to import code at the top level of the module. They are resolved and processed **before the module is executed**.

```
import { someFunction } from './module.js'; // Import named export
import defaultExport from './defaultModule.js'; // Import default export
import * as allExports from './allModule.js'; // Import all exports as object
```

- To run ES modules in Node.js
  - Add the "type":"module" to package.json
  - or
  - Use the .mjs extension

What will be the output?

```
import fs from "fs";

process.nextTick(() => console.log('nextTick 1'));
Promise.resolve().then(() => console.log('promise 1'));
setImmediate(() => { console.log('setImmediate 1') });
setTimeout(() => console.log('setTimeout 1'), 0);
fs.readFile('./files/input.txt', "utf-8", (err, data) => {
    if (err)
        console.log('there is an error. can not read from file');
    else {
        console.log(data);
    }
});
```

What will be the output?

In commonJS module

- **Output is:**

```
nextTick 1
promise 1
setTimeout 1
setImmediate 1
Hello from input.txt file
```

In ES module

- **Output is:**

```
promise 1
nextTick 1
setImmediate 1
setTimeout 1
Hello from input.txt file
```

ES Module loading is asynchronous

- When Node.js loads an ES modules from the file system, it uses the `fs.promises` API internally to read the file asynchronously.
- This ensures non-blocking I/O operations, allowing Node.js to continue handling other tasks while waiting for the file to be read.

```
import fs from 'fs/promises';
try {
    const content = await fs.readFile('./files/moduleA.js', 'utf8');
    console.log('Module content:', content);
} catch (err) {
    console.error('Failed to read module file:', err);
}
```

Type of Modules

- Your own Modules
  - Simply create a normal JS file it will be a module *(without affecting the rest of other JS files and without messing with the global scope).*

- Built-in Modules
  - Node.js has a set of core modules which you can use without any further installation.
  - fs, path, url, http, etc.
  - Built-in Modules Reference

- Third-Party modules on www.npmjs.com
  - Node.js supports **third-party modules** that you can install using the **npm** (Node Package Manager). These modules are published by the community and can be used to extend the functionality of your application.

path and url module

- The path module provides utilities for working file and directory paths.
- It provides a lot of very useful functionality to access and interact with the file system.

```
import url from 'url';
import path from 'path';
import fs from 'fs';

// Convert to a regular file path
const __filename = url.fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// join method joins multiple path segments into a single valid file path
const filePath = path.join(__dirname, 'input.txt');

// Reading from file synchronously
const content = fs.readFileSync(filePath, 'utf8');

console.log('File Content:', content);
```


Using URL Module

The url module provides utilities for manipulating URLs.

```
import { URL } from 'url';

const myUrl = new URL('https://example.com/path?name=John&age=30#section');

// Access URL components
console.log(myUrl.protocol); // Output: 'https:'
console.log(myUrl.host);     // Output: 'example.com'
console.log(myUrl.pathname); // Output: '/path'
console.log(myUrl.search);   // Output: '?name=John&age=30'
console.log(myUrl.hash);     // Output: '#section'

// Add a query parameter
myUrl.searchParams.append('city', 'New York’);

console.log(myUrl.href);
// Output: 'https://example.com/path?name=John&age=30&city=New+York#section'
```


Parsing the Query String

```
import url from 'url';

const adr = 'http://localhost:8080/default.html?year=2017&month=february';
const q = url.parse(adr, true);

console.log(q.host);          //    'localhost: 8080'
console.log(q.pathname);      //    '/default.html'
console.log(q.search);        //    '?year = 2017 & month=february'

let qdata = q.query;          //    { year: 2017, month: 'february' }
console.log(qdata.month);     //    'february'
```

Example Read/Write Files

```
import path from 'path';
import fs from 'fs';
import url from 'url';

let __filename = url.fileURLToPath(import.meta.url);
let __dirname = path.dirname(__filename);

// Reading from a file:
fs.readFile(path.join(__dirname, '/files/input.txt'),
    { encoding: 'utf-8' }, function (err, data) {
        if (err) throw err;
        console.log(data);
    });
// Writing to a file:
fs.writeFile(path.join(__dirname, 'files', 'output.txt'), 'Hello World!', function (err) {
    if (err) throw err;
    console.log('Done');
});
```

What's the problem with the code above?

Streams

- Collection of data that might not be available all at once and don't have to fit in memory. Streaming the data means an application processes the data while it's still receiving it. This is useful for extra large datasets, like video or database migrations.
- Stream types:
  - Readable (fs.createReadStream): a stream you can pipe from, but not pipe into (you can receive data, but not send data to it). When you push data into a readable stream, it is buffered, until a consumer starts to read the data.
  - Writable (fs.createWriteStream): a stream you can pipe into, but not pipe from (you can send data, but not receive from it)
  - Duplex (net.Socket): a stream you can both pipe into and pipe from, basically a combination of a Readable and Writable stream.
  - Transform (zlib.createGzip): a Transform stream is similar to a Duplex, but the output is a transform of its input.

Streams

Readable Streams

- HTTP responses, on the client
- HTTP requests, on the server
- fs read streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdout and stderr
- process.stdin

Writable Streams

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdin
- process.stdout, process.stderr

Streams example

```
import path from 'path';
import fs from 'fs';
import url from 'url';

let __filename = url.fileURLToPath(import.meta.url);
let __dirname = path.dirname(__filename);

// Stream will read the file in chunks
// if file size is bigger than the chunk then it will read a chunk and emit a 'data' event.
// Use highWaterMark to set the size of the chunk
const readable = fs.createReadStream(path.join(__dirname, '/files/input.txt'),
    { highWaterMark: 64 * 1024, encoding: "utf-8" });

const writable = fs.createWriteStream(path.join(__dirname, '/files/destinationFile.txt'));

readable.on('data', function (chunk) {
    console.log(chunk);
    writable.write(chunk);
});
```

Stream - readable

```
import * as fs from 'fs';

// Create a readable stream
const readableStream = fs.createReadStream('example.txt', { encoding: 'utf8' });

// Set up an event listener to handle data chunks
readableStream.on('data', (chunk) => { console.log('Received
chunk:', chunk);
});

// Handle the end of the stream
readableStream.on('end', () => {
  console.log('Finished reading the file.');
});

// Handle errors readableStream.on('error', (err) => {
  console.error('Error reading the file:', err);
});
```

Stream - writable

```
import * as fs from 'fs';

// Create a writable stream
const writableStream = fs.createWriteStream('output.txt');

// Write data to the file

writableStream.write('Hello, Node.js Streams!\n');

writableStream.write('This is another line.\n’);


// End the stream
writableStream.end(() => {
  console.log('Finished writing to the file.');
});

// Handle errors

writableStream.on('error', (err) => {
  console.error('Error writing to the file:', err);
});
```

Stream - pipe

- In Node.js, the `pipe()` method is a convenient way to handle streaming data between streams. It allows you to connect a readable stream to a writable stream, so data flows from the source to the destination seamlessly. This is particularly useful for processing files, HTTP requests, and other data sources.

```
import * as fs from 'fs';

const sourceStream = fs.createReadStream('source.txt', { encoding: 'utf8' });
const destinationStream = fs.createWriteStream('destination.txt');

// Pipe the readable stream to the writable stream
sourceStream.pipe(destinationStream);

destinationStream.on('finish', () => {
    console.log('File copied successfully.');
});

sourceStream.on('error', (err) => {
    console.error('Error reading source file:', err);
});

destinationStream.on('error', (err) => {
    console.error('Error writing destination file:', err);
});
```

Pipes: src.pipe(dst);

- To connect two streams, Node provides a method called pipe() available on all readable streams. Pipes hide the complexity of listening to the stream events.

```
import fs from 'fs';

const readable = fs.createReadStream('./imgs/dog.jpg');
const writable = fs.createWriteStream('./imgs/destinationImage.jpg’);

readable.pipe(writable);


// note that pipe return the destination, this is why you can pipe it again
to another stream if the destination was readable in this case it has to be
DUPLEX because you are going to write to it first, then read it and pipe it
again to another writable stream.
```

Node as a Web Server

- Node started as a Web server and evolved into a much more generalized framework.
- Node `http` module is designed with streaming and low latency in mind.
- Node is very popular today to create and run Web servers.

Web Server Example

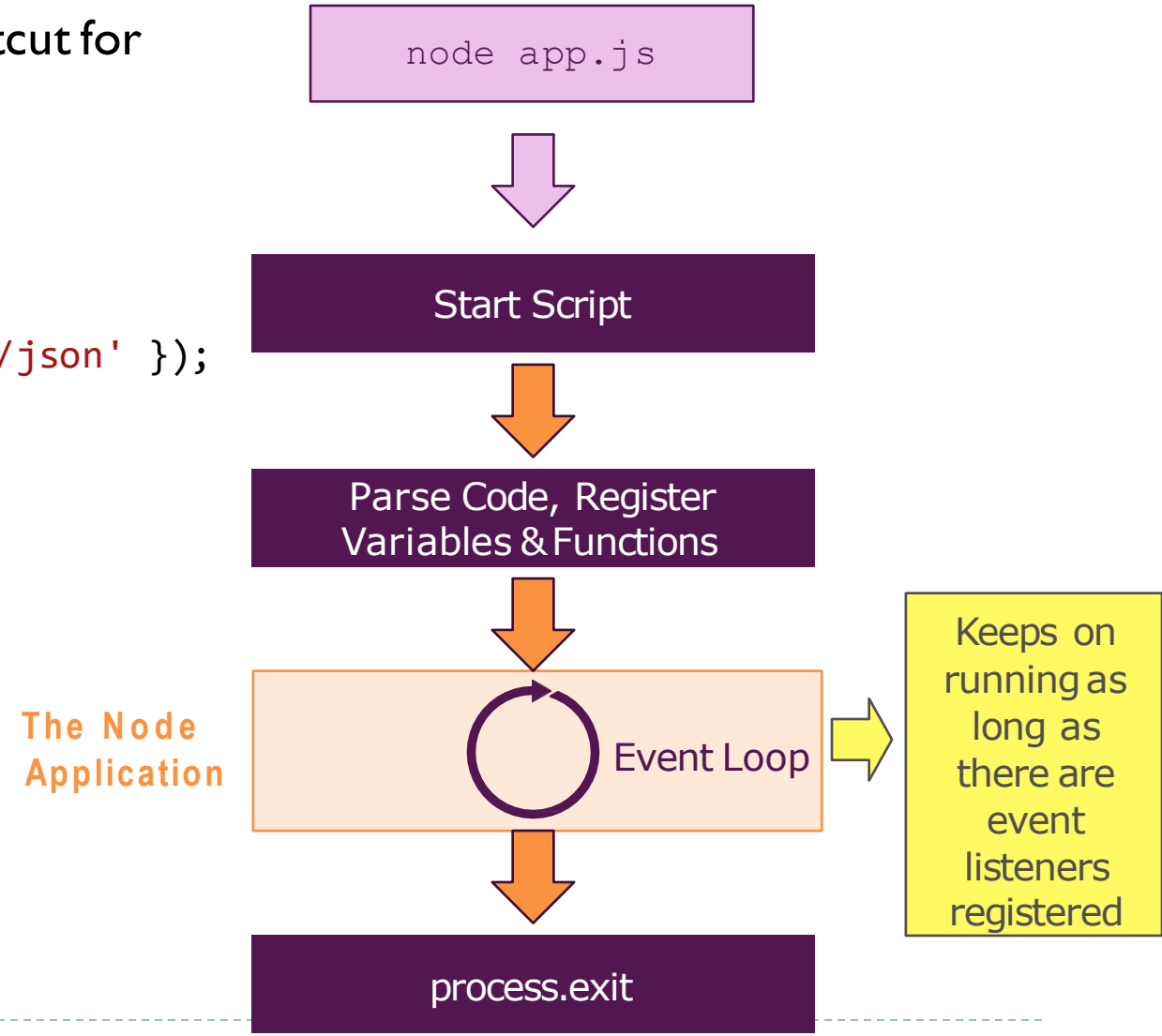
```
import http from "http"
const server = http.createServer();


server.on('request', function (req, res) {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.write('Hello World!');
    res.end();
});

server.listen(3000, () => {
    console.log('Running on port 3000');
});
```

After we run this code. The node program doesn't stop. it keeps waiting for request.

Web Server Example Shortcut

- Passing a callback function to createServer() is a shortcut for listening to "request" event.

```
import http from "http"

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'application/json' });
    const person = {
        firstname: 'Josh',
        lastname: 'Edward'
    };
    res.end(JSON.stringify(person));
}).listen(3000, '127.0.0.1', () => {
    console.log('Running on port 3000');
});
```



Understanding Request & Response

- req and res objects are created by Node.js when a new HTTP request comes in.
  - req – represents the **incoming request** (method, URL, headers, body)
    - an instance of http.IncomingMessage
    - Provides **information about the incoming request**
  - res – represents the **outgoing response** (write headers, body, end)
    - an instance of http.ServerResponse
    - Used to **send data back to the client**

```
import http from 'http';
http.createServer((req, res) => {
    console.log(req.url, req.method, req.headers);
    res.setHeader('Content-Type', 'text/html');
    res.write('My First Page');
    res.write('Hello From Node.js');
    res.end();
}).listen(3000);
```

Send out an HTML file

```
index.html
<html>
        <head></head>
        <body>
                <h1>{Message}</h1>
        </body>
</html>
```

```
import http from "http"
import fs from "fs"

http.createServer(function (req, res) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    var html = fs.readFileSync('./files/index.html', 'utf8');
    var message = 'Hello from Node.js!';
    html = html.replace('{Message}', message);
    res.end(html);
}).listen(1337, '127.0.0.1', () => {
    console.log('Running on port 1337');
});
```

Reading the file

```
import http from "http"
import fs from "fs"

let server = http.createServer();

server.on('request', (req, res) => {
    fs.readFile('./imgs/dog.jpg', (err, data) => {
        if (err) throw err;
        res.end(data);
    });
});

server.listen(8000);
```

A Simpler solution – Use Stream

- We can simply use stream.pipe(), which does exactly what we described.

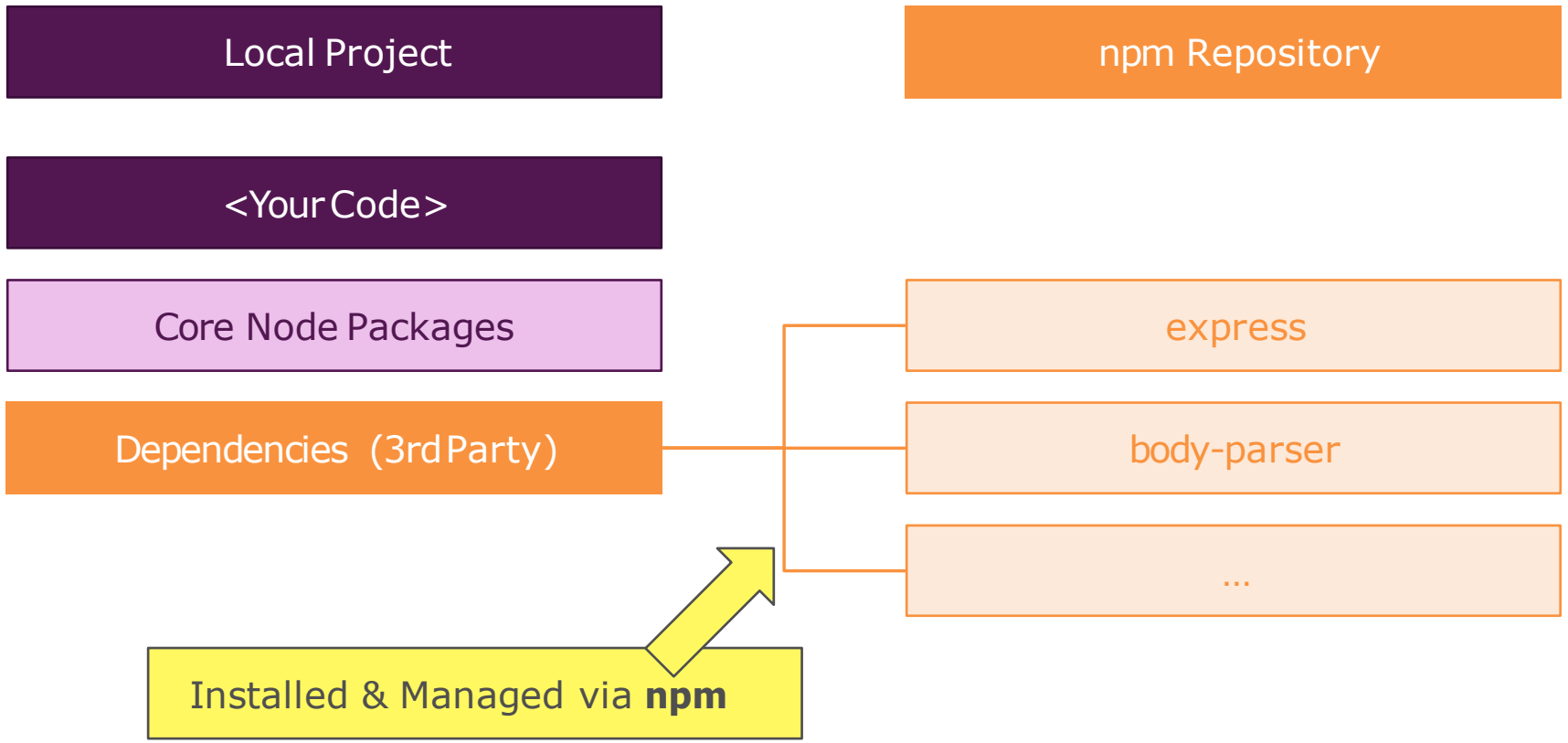
```
import http from "http"
import fs from "fs"

let server = http.createServer();

server.on('request', (req, res) => {
    const src = fs.createReadStream('./imgs/dog.jpg');
    src.pipe(res);
});

server.listen(8000);
```

Routing Requests

```
import http from 'http';

const server = http.createServer((req, res) => {
    const url = req.url;
    const method = req.method;
    if (url === '/') {
        // do something…
    }
    else if (url === '/messsage' && method === 'POST') {
        // do something…
    }
    // do something…
    else if (url === '/messsage' && method === 'GET') {
        // do something…
    }...
    // do something…
});

server.listen(3000);
```

NPM

npm & packages Intro



What is npm?

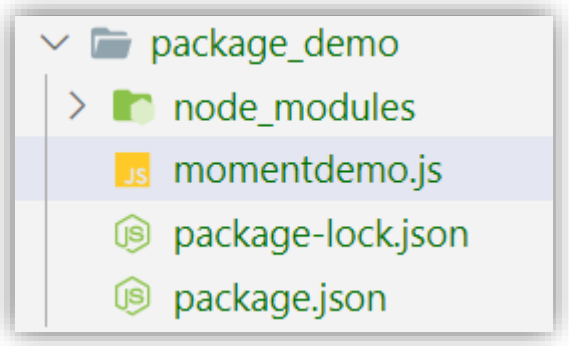
- **npm** is the standard package manager for Node.js. It also manages downloads of dependencies of your project.
- www.npmjs.com hosts thousands of free packages to download and use.
- The NPM program is installed on your computer when you install Node.js.
  - `npm -v // will print npm version`

- What is a package?
  - A package in Node.js contains all the files you need for a module.
  - Modules are JavaScript libraries you can include in your project.

  - A package contains:
    - JS files
    - `package.json` (manifest)
    - `package-lock.json` (maybe)

Create & use a new package

```
npm init // will create package.json
```



- When we install a package:
  - Notice dependencies changes in `package.json`
  - notice folder: `node_modules`
  - This structure separate our app code to the dependencies. Later when we share/deploy our application, there's no need to copy `node_modules`,
    run: `npm install` will read all dependencies and install them locally.

package.json Manifest

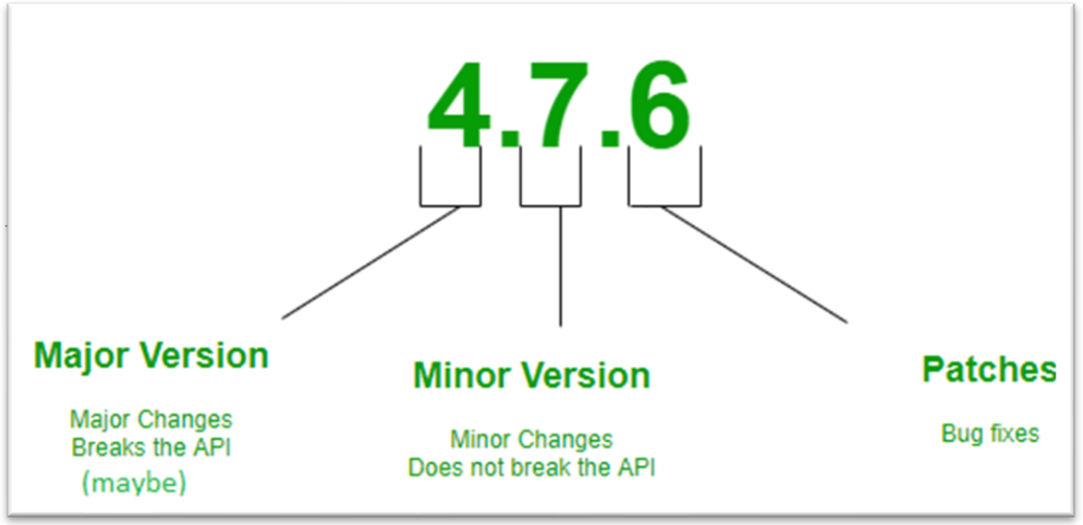
- The `package.json` file is kind of a manifest for your project.
- It can do a lot of things, completely unrelated.
- It's a central repository of configuration for installed packages.
- The only requirement is that it respects the JSON format.

- `version`: indicates the current version
- `name`: the application/package name
- `description`: a brief description of the app/package
- `main`: the entry point for the application
- `scripts`: defines a set of node scripts you can run
- `dependencies`: sets a list of npm packages installed as dependencies
- `devDependencies`: sets a list of npm packages installed as development dependencies

```
{
    "name": "package_demo",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "start": "node momentdemo.js"
     },
    "author": "Anna Smith",
    "license": "ISC",
    "dependencies": {
        "moment": "^2.29.1"
    },
    "devDependencies": {
        "eslint": "^7.28.0"
    }
}
```

Semantic Versioning

- The Semantic Versioning concept is simple: all versions have 3 digits: `x.y.z`.
  - the first digit is the major version
  - the second digit is the minor version
  - the third digit is the patch version



- When you make a new release, you don't just up a number as you please, but you have rules:
  - you up the **major** version when you make incompatible API changes
  - you up the **minor** version when you add functionality in a backward-compatible manner
  - you up the **patch** version when you make backward-compatible bug fixes

More details about Semantic Versioning

- Why is that so important?
  - Because `npm` set some rules we can use in the `package.json` file to choose which versions it can update our packages to, when we run `npm update`.

- The rules use those symbols:
  - **^**: it's ok to automatically update to anything within this major release. If you write `^0.13.0`, when running `npm update`, it can update to `0.13.1`, `0.14.2`, and so on, but not to `1.14.0` or above.
  - **~**: if you write `~0.13.0` when running `npm update` it can update to patch releases: `0.13.1` is ok, but `0.14.0` is not.
  - **>**: you accept any version higher than the one you specify

package-lock.json

- Introduced by NPM version 5 to capture the exact dependency tree installed at any point in time.
- Describes the exact tree
- Guarantee the dependencies on all
- environments.
- Don't modify this file manually.
- Always use npm CLI to change dependencies, it'll automatically update package-lock.json

```
{
  "name": "lesson03-demo",
  "version": "1.0.0",
  "lockfileVersion": 1,
  "requires": true,
  "dependencies": {
    "moment": {
      "version": "2.24.0",
      "resolved": "https://registry.npmjs.org/moment/-
/moment-2.24.0.tgz",
      "integrity": "sha512-
bV7f+6l2QigeBBZSM/6yTNq4P2fNpSWj/0e7jQcy87A8e7o2nAfP/34/2ky5
Vw4B9S446EtIhodAzkFCcR4dQg=="
    }
  }
}
```

More About Packages

- Development Dependencies: Needed only while I'm developing the app. It's not needed for running the app.
  - `npm install mocha --save-dev`

```
// notice devDependencies entry now in package.json
```

- Global Dependencies: Available to all applications
  - `npm install -g nodemon`
  - `nodemon app.js` `//auto detects changes and restarts your project`