FOCUSONFORCE

Server Side JavaScript

Know the core Node.js modules and given requirements, infer which Node.js library/framework is a good solution.

Develop your career with JavaScript Training and Certification Preparation

After studying this topic, you should be able to:

- Identify the three types of modules that can be used in a Node.js application.
- Identify some of the most commonly used Node.js core modules and describe how to use them.
- Describe how to use local modules in Node.js
- Describe how to use third-party modules in Node.js

Introduction

When creating a Node.js application or web server, a developer can make use of three types of modules. These are **core modules**, **local modules**, and **third-party modules**. A core module is built-in and defined in the Node.js source. They do not need to be installed. A local module is created locally by the developer in the application. An example of a local module is a **helper.js** module that contains various utility functions. The require() function is used to include a core, local or third-party module in Node.js.

Using a third-party module has certain advantages. There are hundreds of thousands of open-source modules available for various use cases in the npm registry. They offer predefined code and can be installed easily using the **npm install <package>** command. These open-source modules include popular **JavaScript frameworks and libraries** like React and Express, which offer strong community and good documentation.

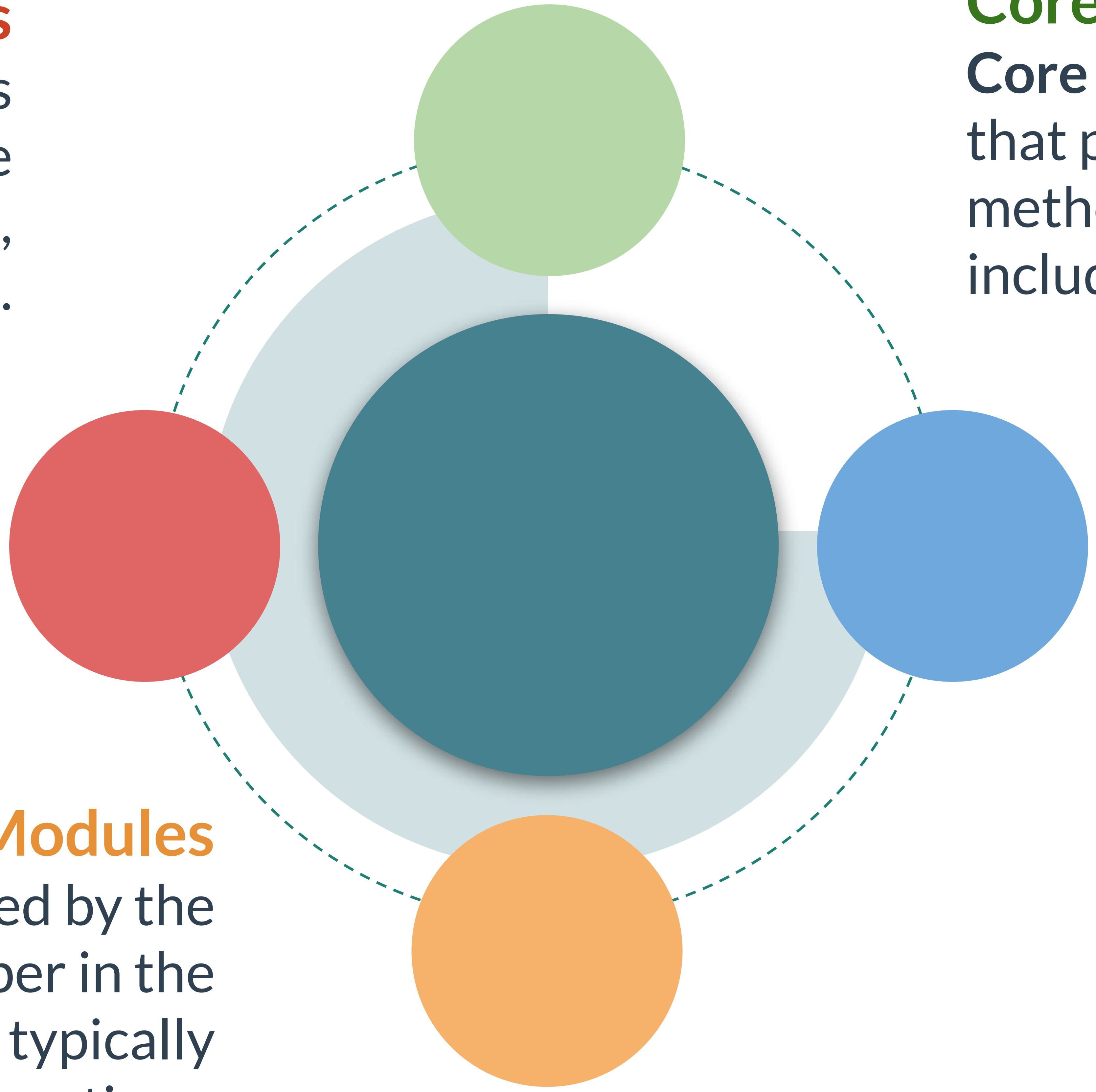
Overview

Node.js Modules

In Node.js, three types of modules can be included and utilized. These are **core modules**, **local modules**, and **third-party modules**.

Core Modules

Core modules are built-in packages that provide various useful classes, methods, and properties. These include http, https, fs, path, etc.



Third-Party Modules

A **third-party module** can be found on the **npm website (npmjs.com)**. It can be installed using the **npm install <package>** command.

Local Modules

A **local module** is defined by the application developer in the application project and typically contains helper functions.

Node.js Modules

A **module** is a package comprising a set of functions that can be included in a Node.js application for a specific use case. There are **three types** of modules:

CORE

Core modules are built-in modules that are defined in the Node.js source. They load automatically when a Node.js application runs and do not need to be installed.

LOCAL

Local modules are created by the developer locally in the application package. They can be packaged and distributed via npm.

THIRD-PARTY

Third-party modules are developed by others and can be accessed via the npm website. Such modules need to be installed before they can be used.

CommonJS Module System

Node.js apps use **CommonJS** for module management. It is a **specification** that defines the behavior of the **exports** variable and the **require** function.

NAMED EXPORTS

To use a **named export**, an **additional property** can be specified on the **exports** object for the function or object that needs to be exported from a module. For example, **exports.user = user;**.

DEFAULT EXPORTS

To use a **default export**, the **module.exports** property can be assigned to a new value, such as a function or object. For example, **module.exports = User;** (where User is a class).

USING EXPORTS

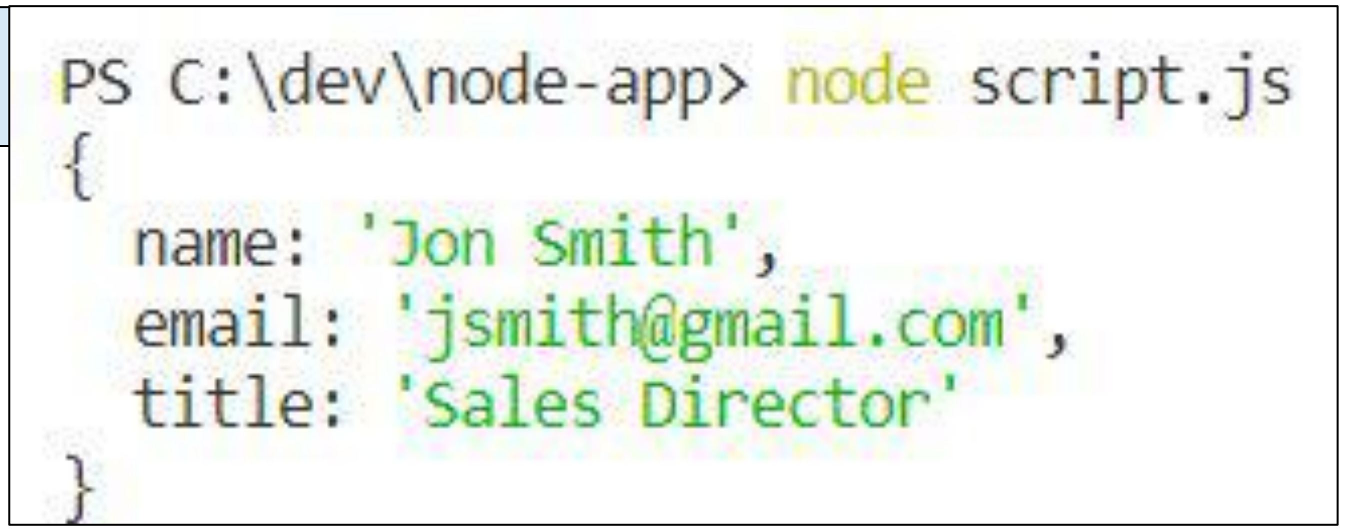
The **require()** function can be added in the main file to use an exported feature. For example **const User = require('./helper.js');** can be added to use the User class that is exported in the helper.js file.

Named Exports

Using the **named export** syntax requires specifying an additional property on the **exports** object for the function or object that needs to be exported.

```
/* In this module named helper.js, an additional property named 'user' has been assigned to the 'exports' object to define a named
export. */
const user = {
    name: 'Jon Smith',
    email: 'jsmith@gmail.com',
    title: 'Sales Director'
};
exports.user = user;

/* In the main JavaScript file named script.js, the exported object is used by utilizing the require() function. */
const helper = require('./helper.js');
const user = helper.user;
console.log(user);
```

Output



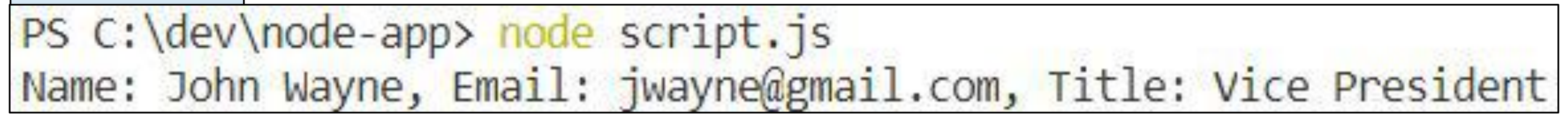
```
PS C:\dev\node-app> node script.js
{
  name: 'Jon Smith',
  email: 'jsmith@gmail.com',
  title: 'Sales Director'
}
```

Default Exports

Using the **default export** syntax requires assigning a new value to the **module.exports** property. It can simplify usage and signal the primary intent of the module.

```
/* In this module named helper.js, a new value has been assigned to the 'module.exports' property. */
class User {
    constructor(name, email, title) {
        this.name = name;
        this.email = email;
        this.title = title;
    }
    displayDetails() {
        console.log(`Name: ${this.name}, Email: ${this.email}, Title: ${this.title}`);
    }
}
module.exports = User;
/* In the main JavaScript file named script.js, the exported class is used by utilizing the require() function. */
const User = require('./helper.js');
const user = new User('John Wayne', 'jwayne@gmail.com', 'Vice President');
console.log(user.displayDetails());
```

Output



```
PS C:\dev\node-app> node script.js
Name: John Wayne, Email: jwayne@gmail.com, Title: Vice President
```

Core Modules

Some of the most important **core modules** in Node.js are as follows:

HTTP/HTTPS

The **http** and **https** modules provide classes, methods, and properties for **creating a web server**, **handling incoming requests**, and **performing outbound requests**.

FS

The **fs** module provides classes and methods for **accessing** and **interacting** with the **file system**. For instance, it can be used to **read** and **edit files** in the application directory.

PATH

The **path** module provides various methods and properties for **working with file** and **directory paths**. For instance, it can be used to **get the last portion of a file path**.

Core Modules

Some of the most important **core modules** in Node.js are as follows:

URL

The **URL** module provides classes, methods, and properties that can be used for **URL resolution and parsing**. For instance, it can be used to get the host name portion of a URL.

EVENTS

The **events** module has classes, methods, and properties which allow **working with events** in Node.js. For instance, it can be used to emit and handle events.

UTIL

The **util** module provides various utilities that are used for **supporting** the **Node.js internal APIs**. For instance, it can be used to check if a value is a Boolean object.

Core Modules

Some of the most important **core modules** in Node.js are as follows:

QUERYSTRING

The **querystring** module gives access to utilities for parsing and formatting URL query strings.For instance, the **querystring.parse()** method parses the provided URL query string into key-value pairs.

ASSERT

The **assert** module provides access to functions that can be used for assertion testing. For instance, **assert.strictEqual()** tests strict equality between the provided actual and expected parameters.

PROCESS

The **process** module can be used to get information about and control the current Node.js process. For instance, **process.exit()** can be used to exit the current process.

Core Modules

The following are also **core modules** in Node.js:

CONSOLE

The **console** module enables a simple debugging console **similar to the JavaScript console** functionality provided by web browsers such as *Google Chrome*.

STREAM

The **stream** module provides an API for implementing the stream interface. A stream instance, for example, is a **request to an HTTP server** or the **process.stdout** property.

OS

The **os** module provides **operating system-related utility methods and properties** such as getting the host name, home directory of the current user, name of the operating system, etc.

Core Modules

The example below makes use of the **path** module to get the directory and last portion of the specified file path.

```javascript
// This example shows how the 'path' module can be used to get information about a file path.
const path = require('path');

const file = './public/img/logo.png';

const directory = path.dirname(file);
const last = path.basename(file);

console.log(`The directory of the file path is ${directory}`);
console.log(`The last portion of the file path is ${last}`);
```

Output

```
The directory of the file path is ./public/img
The last portion of the file path is logo.png
```

Core Modules

The example below makes use of the **util** module to check whether a function is an **async** function.

```javascript
// This example shows how the 'util' module can be used to determine the type of function in Node.js.
const util = require('util');

async function fetchData() {
    // Code that performs a GET request
}

const isAsync = util.types.isAsyncFunction(fetchData);
const message = `The fetchData function is ${isAsync ? 'asynchronous' : 'synchronous'}.`;

console.log(message);
```

Output

```
The fetchData function is asynchronous.
```

Core Modules

The example below makes use of the **events** module to create and handle a custom event.

```
const events = require('events');
const appEvents = new events.EventEmitter();

// Implementation of application
appEvents.on('ping', function(data) {
 console.log(data);
});

// Send the event
appEvents.emit('ping', 'App has pinging.');
```

Output

```
App has pinging.
```

Local Modules

A developer can create a **local module** in a Node.js application for a particular set of functionalities. For example, a helper module can be **created**, **exported** and **included** in the main application file.

```
// The code below is in a local module named helper.js. The getCurrentDate() function has been exported.
function getCurrentDate() {
  const now = new Date();
  const months = ['January','February','March','April','May','June','July','August','September','October','November','December'];
  const currentDate = `${months[now.getMonth()]} ${now.getDate()}, ${now.getFullYear()}`;
  return currentDate;
}
module.exports = getCurrentDate;

// This is the main application file named script.js that includes the local module using require() and uses its function.
const getCurrentDate = require('./helper.js');

const currentDate = getCurrentDate();
console.log(`The current date is ${currentDate}.`);
```

Output

```
The current date is November 3, 2020.
```

Third-Party Modules

Third-party modules can be downloaded and installed via **npm (node package manager)** using the **npm install <package>** command.

NPM WEBSITE

The **npm website (npmjs.com)** can be used to search for third-party modules and access their documentation.

PURPOSE

A third-party module can be installed for the purpose of **development only** or **running the application**.

REQUIRE

Once a third-party module has been installed, it must be loaded using the **require()** function.

Benefits of Third-Party Modules

Some of the most important **benefits** of using third-party npm modules/packages are as follows:

OPEN-SOURCE	PREDEFINED CODE	FRAMEWORKS
There are hundreds of thousands of **open-source packages** that can be installed and used in an application easily using just the command-line interface.	Instead of developing new code, a third-party module that contains **predefined code** can be utilized to meet a particular requirement. For example, a module can be installed for validating input.	**JavaScript frameworks** such as React and Angular can be used to make it easier to develop applications. They offer **strong community** and **good documentation**.

Popular Third-Party Modules

Some of the most **depended** and **popular** third-party npm modules/packages in no particular order are as follows:

LODASH

Lodash is a JavaScript library that makes it easier to work with objects, array,s numbers, strings, etc.

REACT

React is a JavaScript library that is used for building user interfaces by defining components.

EXPRESS

Express is a web framework that allows building a web server and handling incoming HTTP requests via routes.

AXIOS

Axios is an HTTP client that allows making XMLHttpRequests from the browser and http requests from Node.js.

Popular Third-Party Modules

Some of the most **dependend** and **popular** third-party npm modules/packages in no particular order are as follows:

WEBPACK

Webpack can be used to bundle JavaScript modules for usage in the browser and also transform, bundle or package other resources or assets.

CHALK

Chalk is used for styling the messages logged in the terminal. For example, a console.log() statement can display a blue message with a white background.

MOMENT

The **Moment** library can be installed to parse, validate, manipulate, and display dates for various use cases.

JEST

Jest is a testing framework that can be used for creating unit tests for testing JavaScript code.

Using a Third-Party Module

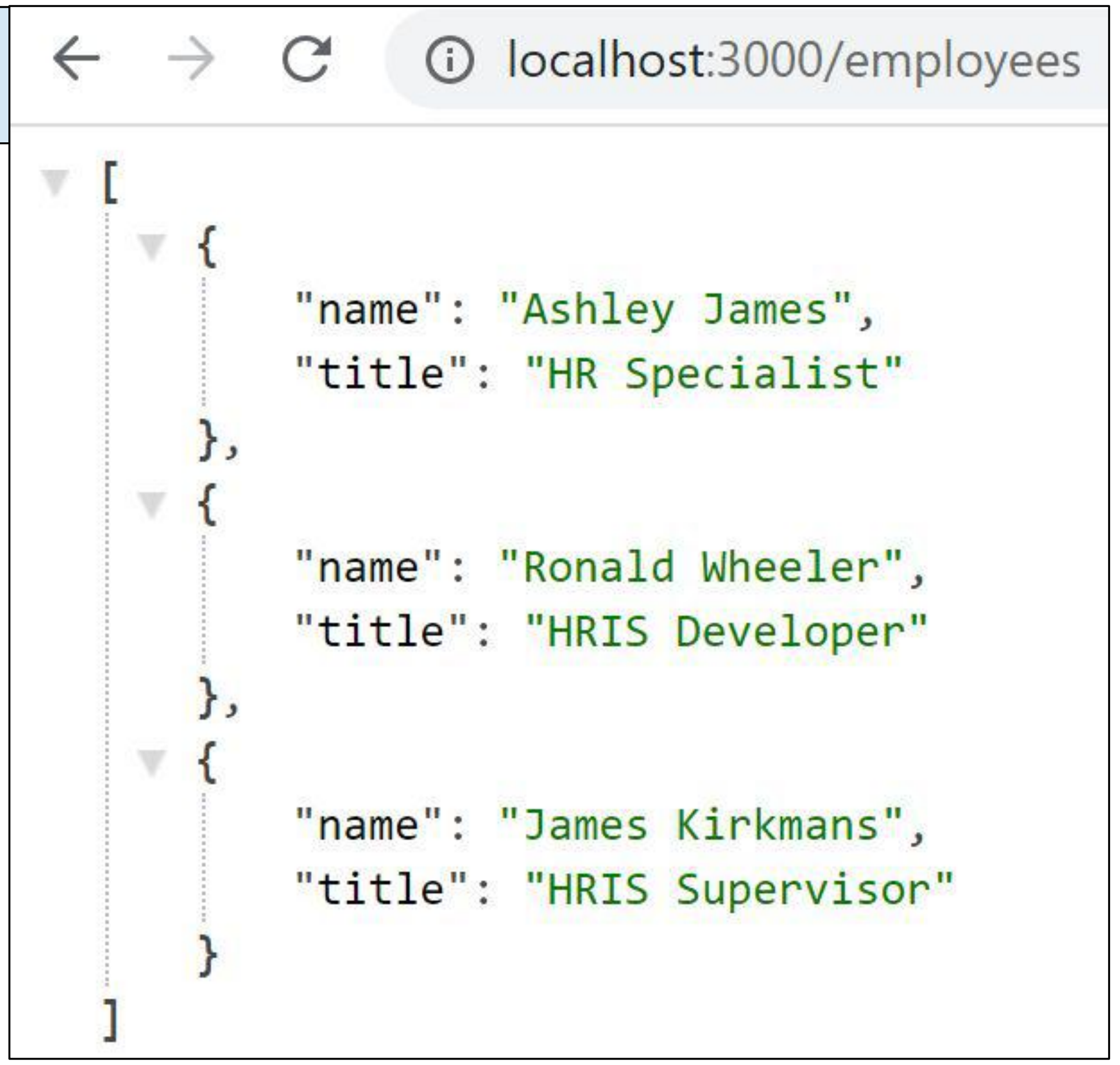
The example below shows how to use the third-party module named **Express** in a Node.js application to create a web server and handle incoming GET requests.

```
// The package must be installed using the 'npm install <package>' command as shown below.
```

```
npm install express
```

```
// In the JavaScript file, the 'express' module must be included using the require() function.
const express = require('express');
const helper = require('./helper.js');
const app = express();
const port = 3000;

app.get('/employees', async (req, res) => {
    const data = await helper.getDataFromDatabase();
    res.send(data);
});

app.listen(port, () => {
    console.log(`App listening at http://localhost:${port}`);
});
```

Output



localhost:3000/employees

```
[
    {
        "name": "Ashley James",
        "title": "HR Specialist"
    },
    {
        "name": "Ronald Wheeler",
        "title": "HRIS Developer"
    },
    {
        "name": "James Kirkmans",
        "title": "HRIS Supervisor"
    }
]
```

Using a Third-Party Module

The example below shows how to use two third-party modules named **Moment** and **Chalk** in a Node.js application to get and display a formatted date that is fourteen days from now.

```
// The two modules must be installed using the 'npm install <package>' command as shown below.
```

```
npm install moment
```

```
npm install chalk
```

```
// In the JavaScript file, 'moment' and 'chalk' must be included using the require() function.
const moment = require('moment');
const chalk = require('chalk');

const date = moment().add('14', 'days').format('MMMM Do, YYYY');
const styledDate = chalk.blue.bgYellowBright.bold(date);

console.log(`Fourteen days from now, the date will be ${styledDate}.`);
```

Output

```
Fourteen days from now, the date will be November 18th, 2020.
```

Learn More

- npmjs
- Most Depended Upon Packages
- Node.js Documentation
- CommonJS Modules

Scenario and Solution

Scenario

A JavaScript developer is building a web service for the HR department of her company using Node.js. She would like to expose the following two endpoints in the web service for incoming GET requests:

1) GET: /employeeId (employeeId will be provided by the user)
2) GET: /employees (a list of HR employees)

She has defined a local module called helper.js which contains two functions named `getEmployee()` and `getEmployeeList()`, one for retrieving the specified HR employee and the other for retrieving the list of HR employees from the database. She needs to use these functions in the main JavaScript file named script.js.

She is also looking for an easier way of setting up the web server and handling the GET requests in Node.js. Furthermore, when the server receives a request containing an employeeId, she would like to display it in the terminal. The employeeId should be displayed in bold, and its color should be cyan. She would like to install and use third-party modules for these requirements.

Solution

The developer can install and use two third-party modules named ‘express’ and ‘chalk’ for these requirements. These third-party modules must be installed using the npm install <package> command in the terminal. Once they have been installed, they can be included with the local module named ‘helper.js’ in the main JavaScript file. The functions in ‘helper.js’ must be exported using a ‘module.exports’ statement.

```
/* The 'express' and 'chalk' modules must be installed using these commands before they can be used.
```

```
npm install express
```

```
npm install chalk
```

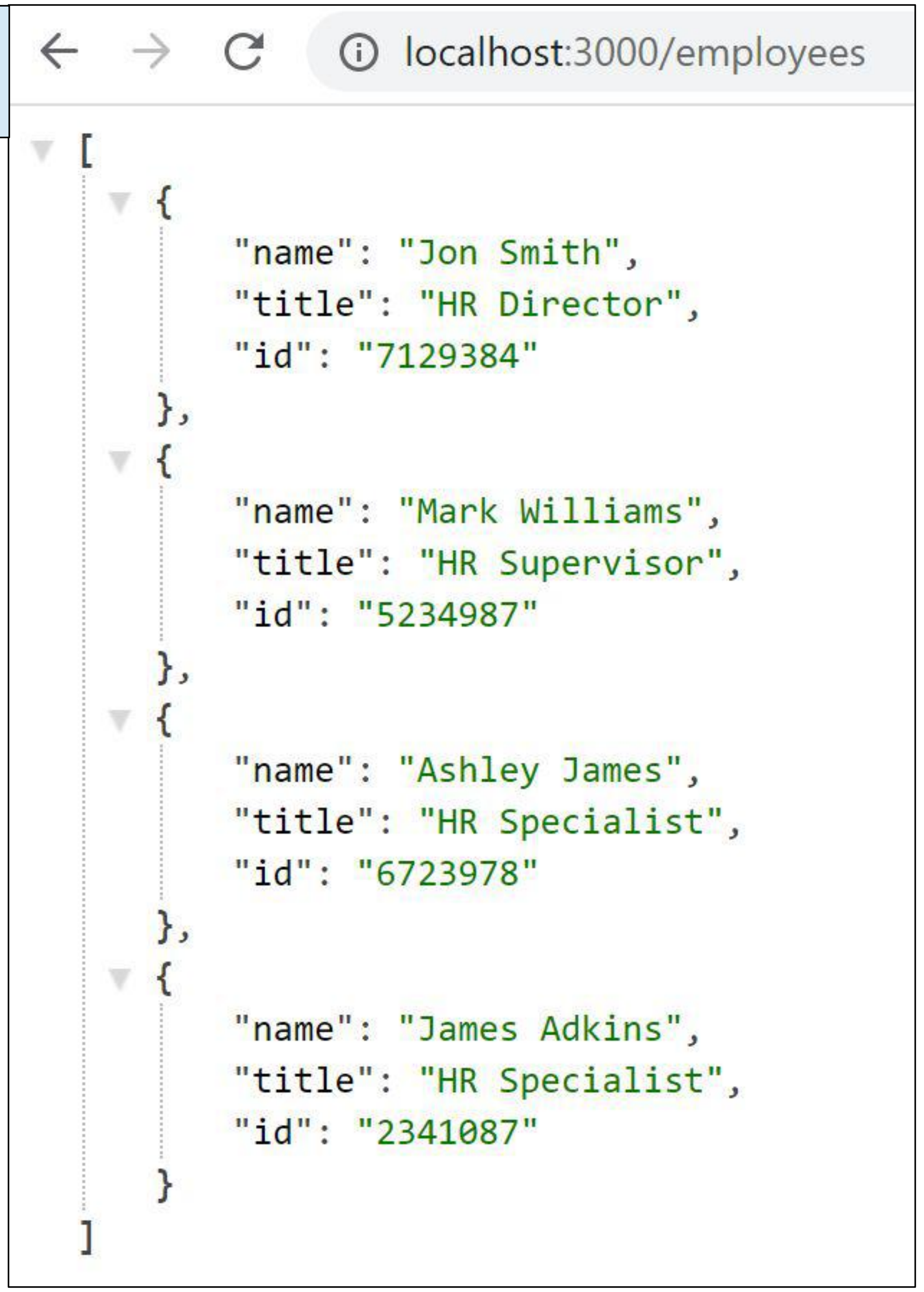
```
/* The local module (helper.js) and the two third-party modules ('express' and 'chalk') must be included in the main
JavaScript file. */
const express = require('express');
const chalk = require('chalk');
const helper = require('./helper.js');
const app = express();
const port = 3000;

// The route for the '/employees' endpoint must be defined first.
app.get('/employees', async (req, res) => {
    const data = await helper.getEmployeeList();
    res.send(data);
});
```

Continued...

Solution

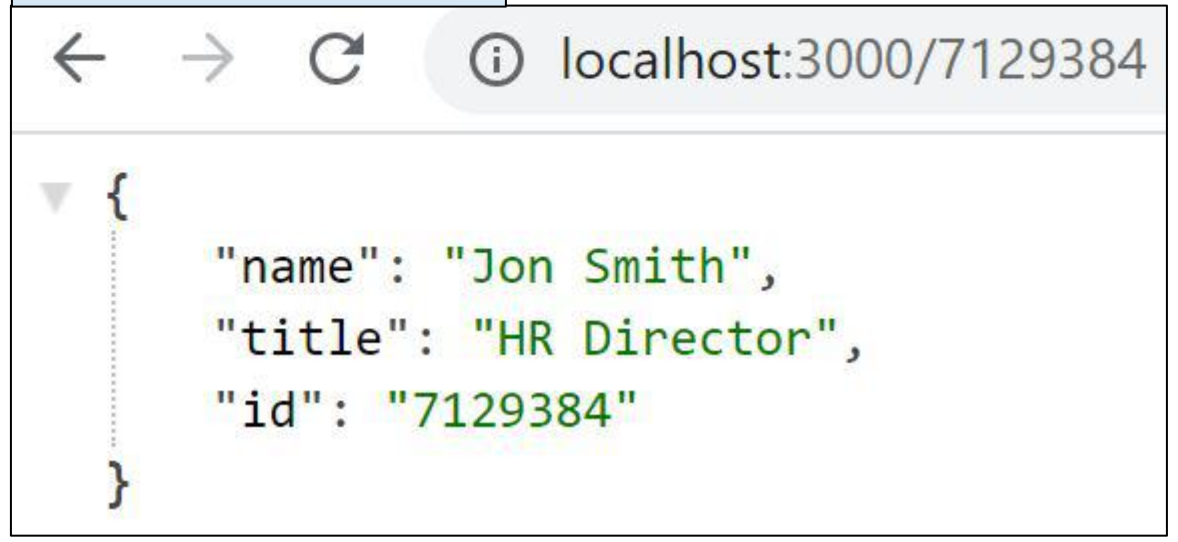
```
// The route for the '/employeeId' endpoint must be defined using a colon (:).
app.get('/:employeeId', async (req, res) => {
    // The employeeId can be accessed using req.params.
    const employeeId = req.params.employeeId;
    // The 'chalk' module can be used to style the employeeId.
    console.log(chalk.cyan.bold(employeeId));
    const data = await helper.getEmployee(employeeId);
    res.send(data);
});

app.listen(port, () => {
    console.log(`App listening at http://localhost:${port}`);
});
```

Browser Output



```
[
  {
    "name": "Jon Smith",
    "title": "HR Director",
    "id": "7129384"
  },
  {
    "name": "Mark Williams",
    "title": "HR Supervisor",
    "id": "5234987"
  },
  {
    "name": "Ashley James",
    "title": "HR Specialist",
    "id": "6723978"
  },
  {
    "name": "James Adkins",
    "title": "HR Specialist",
    "id": "2341087"
  }
]
```

Browser Output



```
{
  "name": "Jon Smith",
  "title": "HR Director",
  "id": "7129384"
}
```

Terminal Output

```
7129384
```