COMP-308 Winter 2018

Introduction to Node.js

Objectives:

- Define **JavaScript closures** and event-driven programming with Node.js.
- Describe **Connect** web framework and Connect's middleware pattern.
- Define **CommonJS** modules and the Node.js module system
- Introduction to the **Connect web framework**
- Connect's **middleware pattern**
- Create Node.js applications using **Connect middleware**

Intro to Node.js

- In 2009, Ryan Dahl presented his project named Node.js.
  - A platform capable of running **complex JavaScript applications** that were **simple** to code, highly **efficient**, and easily **scalable**.
- Node.js uses the **event-driven nature of JavaScript to support non-blocking operations** in the platform, a feature that enables its excellent efficiency.

JavaScript event-driven programming

- Synchronous programming example:

```
System.out.print("What is your name?");
String name = System.console().readLine();
System.out.print("Your name is: " + name);
```

  - The program executes the first and second lines, but any code after the **second line will not be executed until the user inputs their name**.
  - This is synchronous programming, where **I/O operations block the rest of the program from running**.
- JavaScript is an **event-driven language**, which means that you **register code to specific events**, and that **code will be executed once the event is emitted**.

JavaScript event-driven programming

- **I/O in JavaScript is non-blocking** - asynchronous I/O implementation
  - when an I/O operation (communicating with a database server, etc) needs to be performed, call the operation asynchronously.
  - The rest of operations will not be blocked.
- Event loop - the mechanism through which the JavaScript runtime handles function **calls**, **events** and **callback** functions
  - **call stack** to manage function calls (**LIFO**)
  - a **message queue** for handling events
  - JavaScript is **non-blocking**, **single-threaded**:
    - once a function starts executing, nothing can interrupt it.
    - Any **callback functions which are ready to be executed have to wait**.

JavaScript event-driven programming

- JavaScript event-driven example:

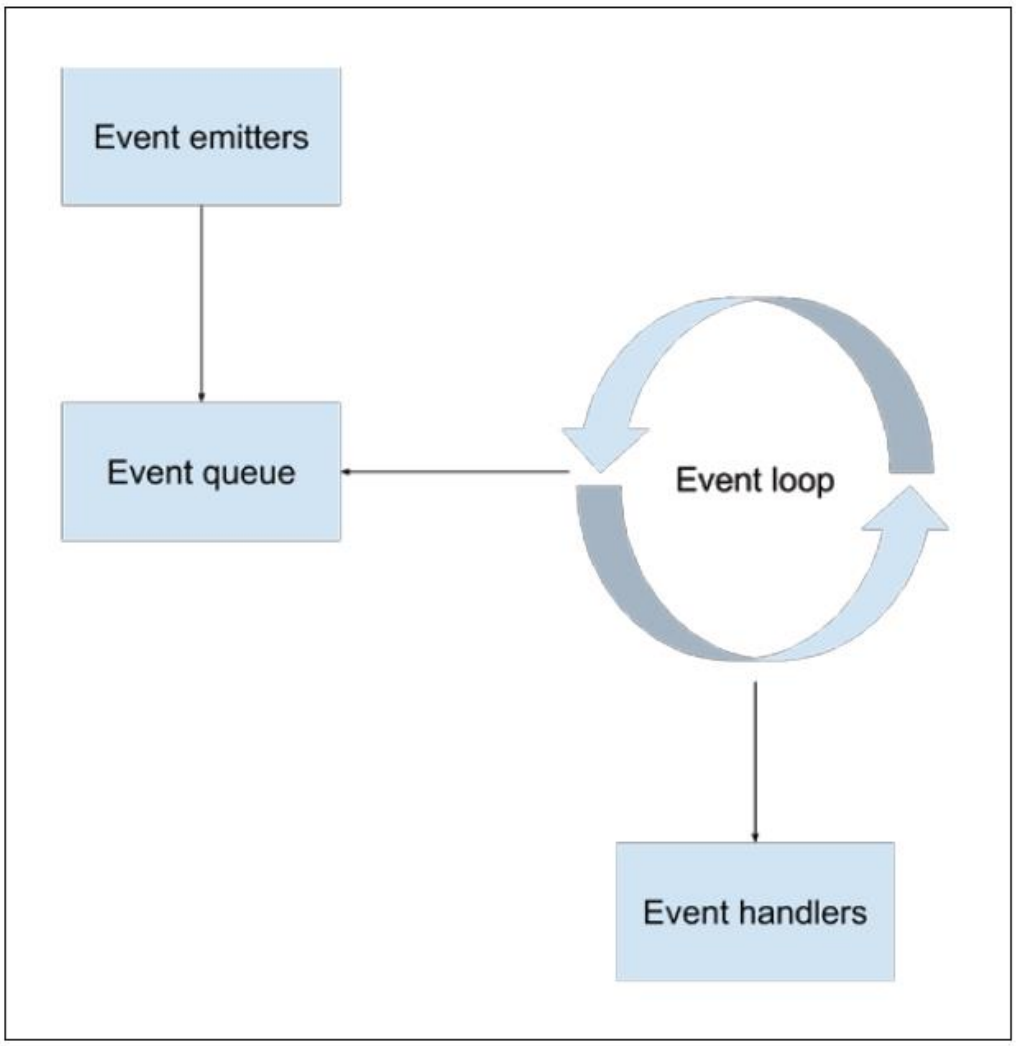
```
<span>What is your name?</span>
<input type="text" id="nameInput">
<input type="button" id="showNameButton" value="Show Name">
<script type="text/javascript">
var showNameButton = document.getElementById('showNameButton');
showNameButton.addEventListener('click', function() {
        alert(document.getElementById('nameInput').value);
});
// Rest of your code...
</script>
```

- The **anonymous function** will **run once the event is emitted**.
- We usually refer to this function as a **callback function**.
- Any code after the **addEventListener()** method will execute accordingly regardless of what we write in the callback function.

JavaScript event-driven programming

- The browser manages a single thread to run the entire JavaScript code using an inner loop, commonly referred to as the **event loop**.
  - The event loop is a **single-threaded loop** that the browser runs infinitely.
  - Every time an **event is emitted**, the browser adds it to an **event queue**.
  - The loop will then grab the next event from the queue in order to **execute the event handlers** registered to that event.
  - After all of the **event handlers are executed**, the loop grabs the next event, executes its handlers, grabs the next event, and so on.

JavaScript event-driven programming



The event loop cycle

Node.js event-driven programming

- When developing web server logic, you will probably notice a lot of your **system resources are wasted on blocking code**.
- Consider the following PHP code that interacts with a MySQL database:

```
$output = mysql_query('SELECT * FROM Users');
echo($output);
```

  - The server will send the query to the database
  - The database will execute the select statement
  - The database will return the result to the PHP code
  - PHP code will output the result.
- The preceding code **blocks any other operation until it gets the result from the database**.

Node.js event-driven programming

- To solve this issue, many web platforms have implemented **a thread pool system** that usually issues **a single thread per connection**.
- This kind of multithreading may seem intuitive at first, but has some **significant disadvantages**, as follows:
  - Managing threads becomes a **complex task**
  - System **resources are wasted** on idle threads
  - **Scaling** these kinds of applications **cannot be done easily**

Node.js event-driven programming

- Using **event-driven architecture** will help you dramatically **reduce the load on your server** while leveraging **JavaScript's asynchronous behavior** in building your web application.
- This approach is made possible thanks to a simple **design pattern**, which is called ***closure*** by JavaScript developers.
- A **closure** is *a function having access to the parent scope, even after the parent function has closed*.

JavaScript closures

- Closures are **functions that refer to variables from their parent environment**.
- Using the closure pattern enables **variables from the parent() function to remain bound to the closure**.
- In the following example the child() function has access to variable *message* defined in the parent() function:

```
function parent() {
    const message = "Hello World";
    function child() {
      alert (message); //has access to parent scope
    }
    child();
}
parent(); //displays Hello World
```

JavaScript closures

- A more interesting example:

```
function parent() {
    const message = 'Hello World';
    function child() {
      alert (message);
    }
    return child;
}
const childFN = parent(); //returns the child() function
childFN(); //displays Hello World
```

- The parent() function returned the child() function, and the **child() function is called after the parent() function has already been executed**.

JavaScript closures

- Closures are very important in asynchronous programming because JavaScript functions are **first-class objects that can be passed as arguments to other functions**.
- This means that:
  - you can **create a callback function and pass it as an argument to an event handler**.
  - When the event occurs, the function will be invoked, and it *will be able to manipulate any variable that existed when the callback function was created even if its parent function was already executed*.
- Using the closure pattern will help you **utilize event-driven programming without the need to pass the scope state to the event handler**.

Node modules

- One of its major design flaws of JavaScript is the **sharing of a single global namespace**.
  - In the browser, when you load a script into your web page, the engine will inject its code into an **address space that is shared by all the other scripts**.
  - This means that when you assign a variable in one script, you **can accidently overwrite another variable** already defined in a previous script.
- **CommonJS modules solves this problem**

CommonJS modules

- The CommonJS standards specify the following **three key components when working with modules**:
  - **require()**: This method is **used to load the module** into your code.
  - **exports**: This object is contained in each module and allows you to **expose pieces of your code** when the module is loaded.
  - **module**: This object was originally used to provide **metadata information about the module**. It also contains the pointer of an **exports object as a property**.
- However, the popular implementation of the *exports* object as a standalone object literally changed the use case of the *module* object.

CommonJS modules

- In Node's CommonJS module implementation, each module is written in a single JavaScript file and has **an isolated scope that holds its own variables**.
- The author of the module can expose any functionality through the *exports* object.
- To understand it better, let's say we created a **module file named *hello.js*** that contains the following code snippet:

```
const message = 'Hello';
exports.sayHello = function(){
    console.log(message);
}
```

CommonJS modules

- Also, let's say we created an **application file named server.js**, which contains the following lines of code:

```
const hello = require('./hello');
hello.sayHello();
```

- In the preceding example, you have the ***hello*** module, which contains a variable named *message*.
- The message variable is self-contained in the *hello* module, which only exposes the *sayHello()* method by defining it as a property of the **exports** object.
- Then, the **application file loads the *hello* module** using the **require()** method, which will allow it to **call the sayHello()** method of the **hello module**.

CommonJS modules

- A **different approach to creating modules** is exposing a single function using the **module.exports** pointer.
- To understand this better, let's revise the preceding example.
- A modified **hello.js** file should look as follows:

```
module.exports = function() {
    const message = 'Hello';
    console.log(message);
}
```

- Then, the module is loaded in the **server.js** file as follows:

```
const hello = require('./hello');
hello();
```

- The application file **uses the *hello* module directly as a function** instead of using the sayHello() method as a property of the *hello* module

Node.js core modules

- The core modules provide most of the basic functionalities of Node, including filesystem access, HTTP and HTTPS interfaces, and much more.
- To **load a core module**, you just need to use the *require* method in your JavaScript file.
- An example code, using the ***fs*** core module to read the content of the environment hosts file, would look like the following code snippet:

```
const fs = require('fs');
fs.readFile('/etc/hosts', 'utf8', (err, data) => {
    if (err) {
    return console.log(err);
    }
    console.log(data);
});
```

Node.js third-party modules

- Use NPM to install third-party modules.
  - NPM installs these modules in a folder named *node_modules* under the root folder of your application.
- To use third-party modules, you can just **require** them as you would normally require a core module.
- Node will first look for the module in the core modules folder and then try to load the module from the module folder inside the *node_modules* folder.
- For instance, to use the express module, your code should look like the following code snippet:

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

Node.js file modules

- In previous examples, you saw how Node loads modules directly from files.
- These examples describe a scenario where the files reside in the same folder.
- However, you can also place your modules inside a folder and load them by **providing the folder path**.
- For instance, if you move your *hello* module to a *modules* folder, the application file would have to change, asking Node to look for the module in the new relative path:

```
const hello = require('./modules/hello');
```

- Note that the path **can also be an absolute path**, as follows:

```
const hello = require('/home/projects/first-example/modules/hello');
```

  - Node will then look for the *hello* module in that path.

Node.js folder modules

- Node also supports the **loading of folder modules**.
- Requiring folder modules is done in the same way as file modules, as follows:

```
const hello = require('./modules/hello');
```

- Now, if a folder named *hello* exists, Node will go through that folder looking for a **package.json** file.
- If Node finds a **package.json** file like the one below, it will try parsing it, looking for the *main* property.

```
{
"name" : "hello",
"version" : "1.0.0",
"main" : "./hello-module.js"
}
```

Node.js folder modules

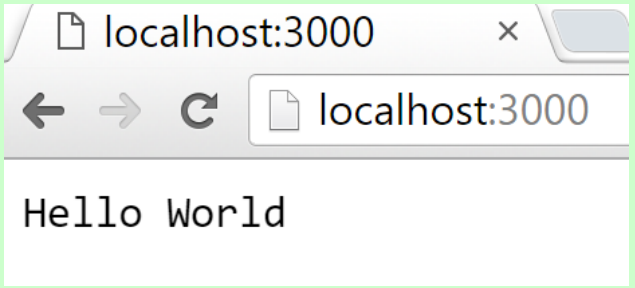
- Node will try to load the **./hello/hello-module.js** file.
- If the **package.json** file doesn't exist or the main property isn't defined, Node will automatically try to load the **./hello/index.js** file.

Developing Node.js web applications

- There are many modules to support web application development but none as popular as the **Connect module**.
- The Connect module delivers **a set of wrappers around the Node.js low-level APIs** to enable the development of rich web application frameworks.
- To understand what Connect is all about, let's begin with a basic **example of a basic Node web server where low-level APIs are used**.
- In your working folder, create a file named **server.js**, which contains the following code snippet:

Developing Node.js web applications

```
const http = require('http');
http.createServer( (req, res)=> {
      res.writeHead(200, {
      'Content-Type': 'text/plain'
});
res.end('Hello World');
}).listen(3000);
console.log('Server running at http://localhost:3000/');
```



- To start your web server, use your command-line tool, and navigate to your working folder.
- Then, run the node CLI tool and run the server.js file as follows: **node server**
- Now open **http://localhost:3000** in your browser, and you'll see the **Hello World** response.

Developing Node.js web applications

- In this example, the **http module is used to create a small web server listening to the 3000 port**.
- You begin by requiring the http module and use the **createServer()** method **to return a new server object**.
- The **listen() method** is then used **to listen to the 3000 port**.
- Notice the **callback function that is passed as an argument to the createServer()** method.
- The callback function **gets called whenever there's an HTTP request** sent to the web server.
- The **server object will then pass the *req* and *res* arguments**, which contain the information and functionality needed to send back an HTTP response

The Connect module

- ***Connect*** *is an extensible HTTP server framework for node, providing high performance "plugins" known as middleware.*
- It wraps the **Server**, **ServerRequest**, and **ServerResponse** objects of node.js' standard http module, giving them a few nice extra features, one of which is allowing the Server object to use a stack of middleware.
- Connect middleware are basically **callback functions**, which get **executed when an HTTP request occurs**.
- The middleware can then **perform some logic**, **return a response**, or **call the next registered middleware**.
- While you will mostly write custom middleware to support your application needs, **Connect** also includes some common middleware to **support logging**, **static file serving**, **and more**.
- The way a Connect application works is by using an object called **dispatcher**.
- The dispatcher object handles each HTTP request received by the server and then decides, in a cascading way, the order of middleware execution.

The Connect module

- Connect isn't a core module, so you'll have to install it using NPM:

npm install connect

- NPM will install the connect module inside a **node_modules** folder, which will enable you to require it in your application file.
- To run your Connect web server, just use Node's CLI and execute the following command:

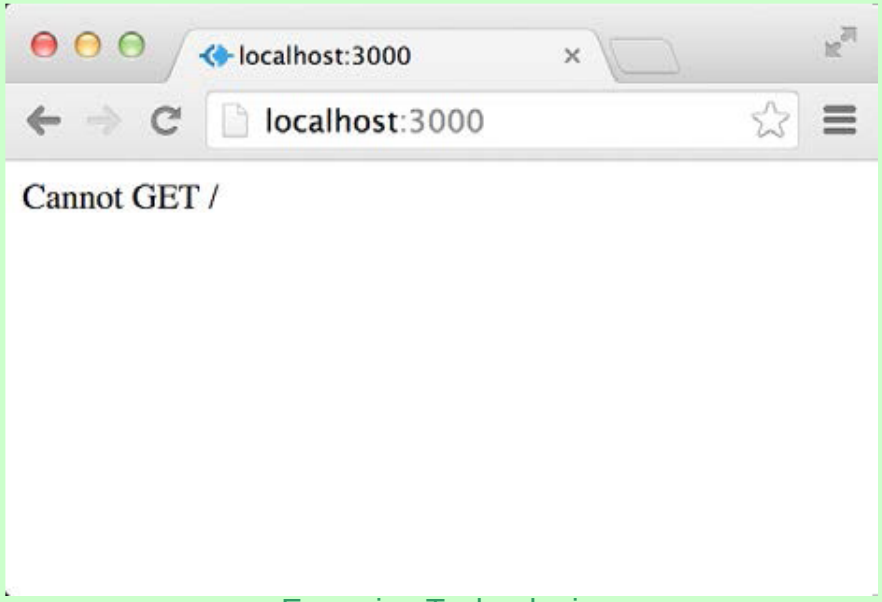
node server

- To create a Connect application create a file named *server.js* in working folder:

```
const connect = require('connect');
const app = connect();
app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

The Connect module

- As you can see, your application file is using the connect module to create a new web server.
- Node will run your application, reporting the server status using the console.log() method.
- The response means is that there isn't any middleware registered to handle the GET HTTP request.



The Connect middleware

- Each middleware function is defined with the following three arguments:
  - **req**: an object that holds the HTTP request information
  - **res**: an object that holds the HTTP response information and allows you to set the response properties
  - **next**: the next middleware function defined in the ordered set of Connect middleware.

The Connect middleware

- Register the middleware with the Connect application using the app.use() method:

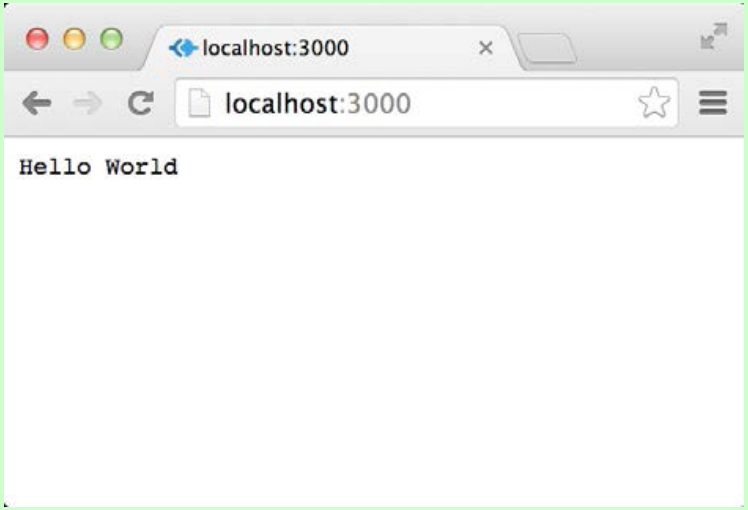
```
const connect = require('connect');
const app = connect();
const helloWorld = function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World');
};
app.use(helloWorld);
app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

- Start your connect server again by issuing the following command in your command-line tool:

node server

The Connect middleware

- Visit http://localhost:3000 again. You will now get a response similar to that in the following screenshot



- First, you added a **middleware function named helloWorld(), which has three arguments: req, res, and next**.
- In your middleware, you used the **res.setHeader() method to set the response Content-Type header** and the **res.end() method to set the response text**.
- Finally, you used the **app.use() method to register your middleware with the Connect application**.

Understanding the order of Connect middleware

- Using the app.use() method, you'll be able to set a series of middleware functions that will be executed in a row to achieve maximum flexibility when writing your application.
- Connect will then pass the next middleware function to the currently executing middleware function using the *next* argument.
- In each middleware function, you can decide whether to call the next middleware function or stop at the current one.
- Each Connect middleware function will be executed in **first-in-first-out** (**FIFO**) order using the *next* arguments until there are no more middleware functions to execute or the next middleware function is not called.

Understanding the order of Connect middleware

```
const connect = require('connect');
const app = connect();
const logger = function(req, res, next) {
    console.log(req.method, req.url);
    next();
};
const helloWorld = function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World');
};
app.use(logger);
app.use(helloWorld);
app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

- The logger() middleware uses the console.log() method to simply log the request information to the console.

Mounting Connect middleware

- Mounting enables you to determine **which request path is required** for the middleware function to get executed.
- Mounting is done by **adding the *path* argument to the app.use() method**:

```
const connect = require('connect');
const app = connect();
const logger = function(req, res, next) {
    console.log(req.method, req.url);
    next();
};
const helloWorld = function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World');
};
```

Mounting Connect middleware

```
const goodbyeWorld = function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Goodbye World');
};
app.use(logger);
app.use('/hello', helloWorld);
app.use('/goodbye', goodbyeWorld);
app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

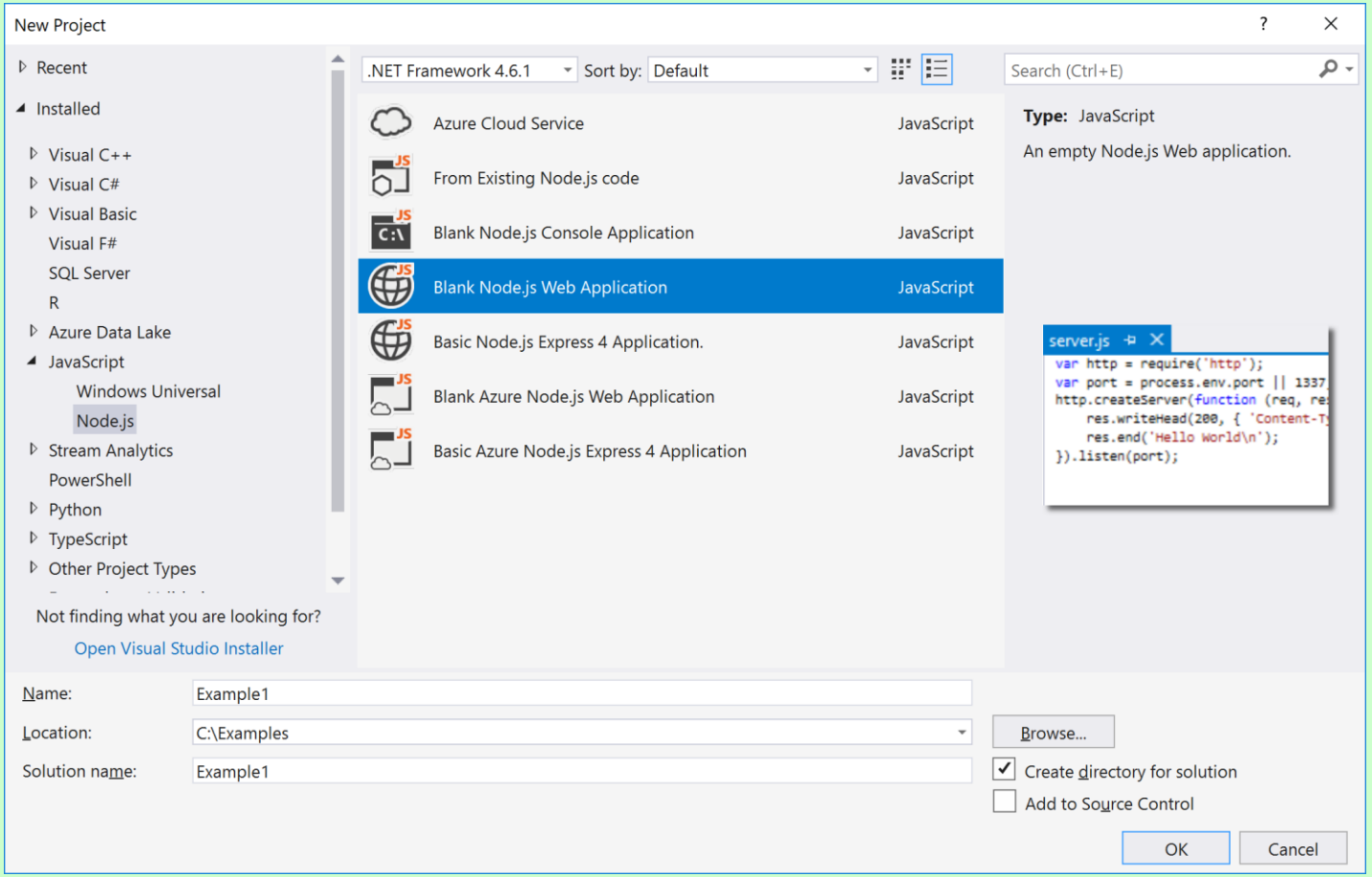
- First, you mounted the helloWorld() middleware to respond only to requests made to the */hello* path.
- Then, you added another middleware called goodbyeWorld() that will respond to requests made to the */goodbye* path.

Connect middleware

- While Connect is a great improvement over writing your web application infrastructure, it deliberately lacks some basic features you're used to having in other web frameworks.
- TJ Holowaychuk, did it better than most when he released a **Connect-based web framework known as Express**.

Connect Examples

- Using VS 2017
  - Create a blank node.js web app:



Connect Examples

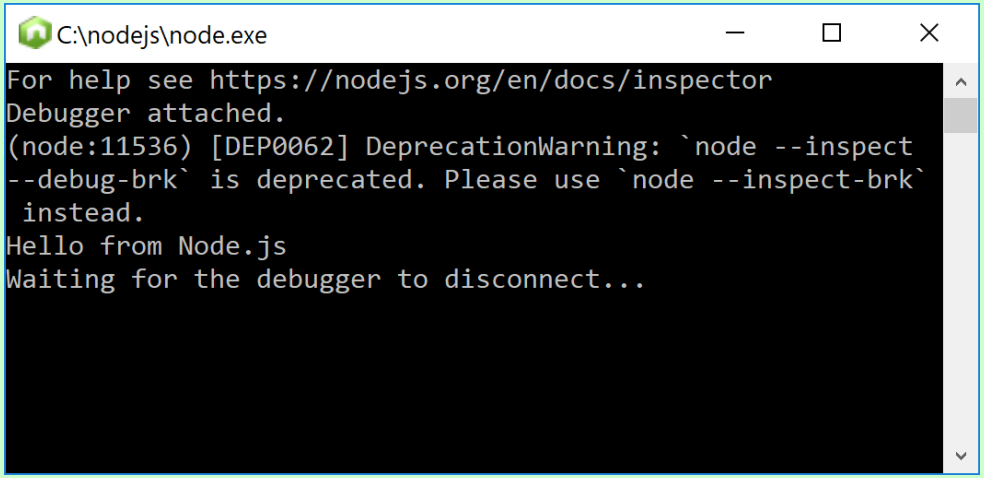
- Create hello.js module:

```
// Define a module variable
const message = 'Hello from Node.js';
// Print message to the console
exports.sayHello = function() {
console.log(message);
};
```

- Use the hello.js module in server.js:

```
// Define a module variable
const message = 'Hello from Node.js';
// Print message to the console
exports.sayHello = function() {
console.log(message);
};
```

- Run server.js



References

- Textbook
- http://www.w3schools.com/js/js_function_closures.asp
- http://code.tutsplus.com/tutorials/meet-the-connect-framework--net-31220
- http://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html
- https://www.visualstudio.com/vs/node-js/
- https://github.com/Microsoft/nodejstools/wiki/Interactive-Window-(REPL)