Collection of JavaScript based technologies used to develop web applications.

node JS™

express

Laiba Imran

What is node.js ?

- An open-source server environment
  - Uses JavaScript on the server
  - Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
  - Runs on Google's V8 JavaScript Engine
- Support **Asynchronous** Programming

Synchronous Programming

- Some tasks may take a very long time, e.g., read/write a file on disk, request data from a server, query a database …
- *Why waiting is bad??*

```
some_task(); /*
  wait for task to
  complete
*/

// process the result
// of task
… …

// do other things
… …
```

Multi-processing and Multi-threading

One process/thread

```
some_task();

// process the
result
// of task
… …
```

Another process/thread

```
// do other things
… …
```

- What's the different between a process and a thread??

Problems of Multi-processing and Multi-threading

- OS must allocate some resources for each process/thread
- Switching between processes and threads (a.k.a. *context switch*) takes time
- Communicating among processes and synchronizing multiple threads are difficult

Big problems for busy web servers

⬇

One of the reasons why Node.js became popular in server-side development

Asynchronous Programming

- Example : Web server
- Open a file on the server and return the content to the client

PHP, ASP, Others

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request

Node

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

… Asynchronous Programming

- Everything runs in one thread
- Asynchronous calls return immediately (a.k.a. *non-blocking*)
- A callback function is called when the result is ready

```
func(args…, callback(err,result))
```

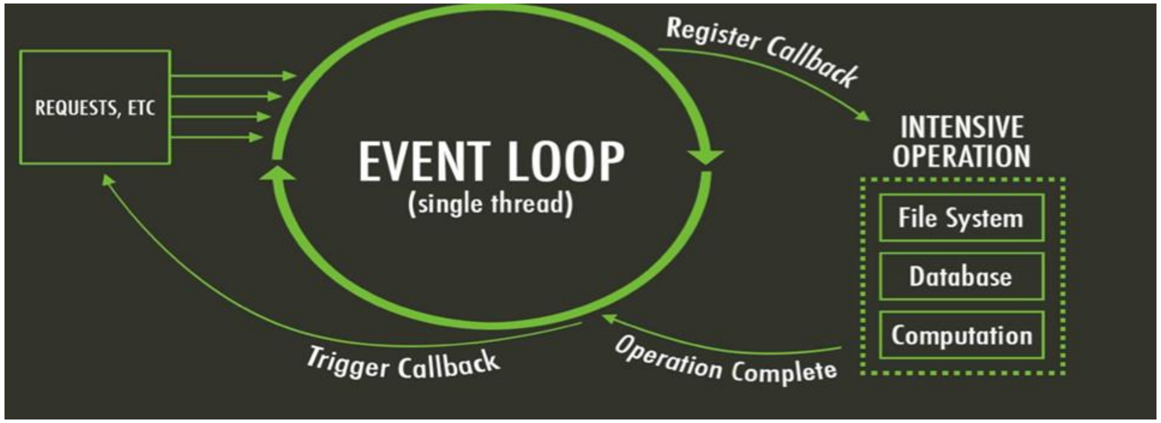
- A.K.A. Event-driven Programming
  - A callback function is basically an event handler that handles the "result is ready" event

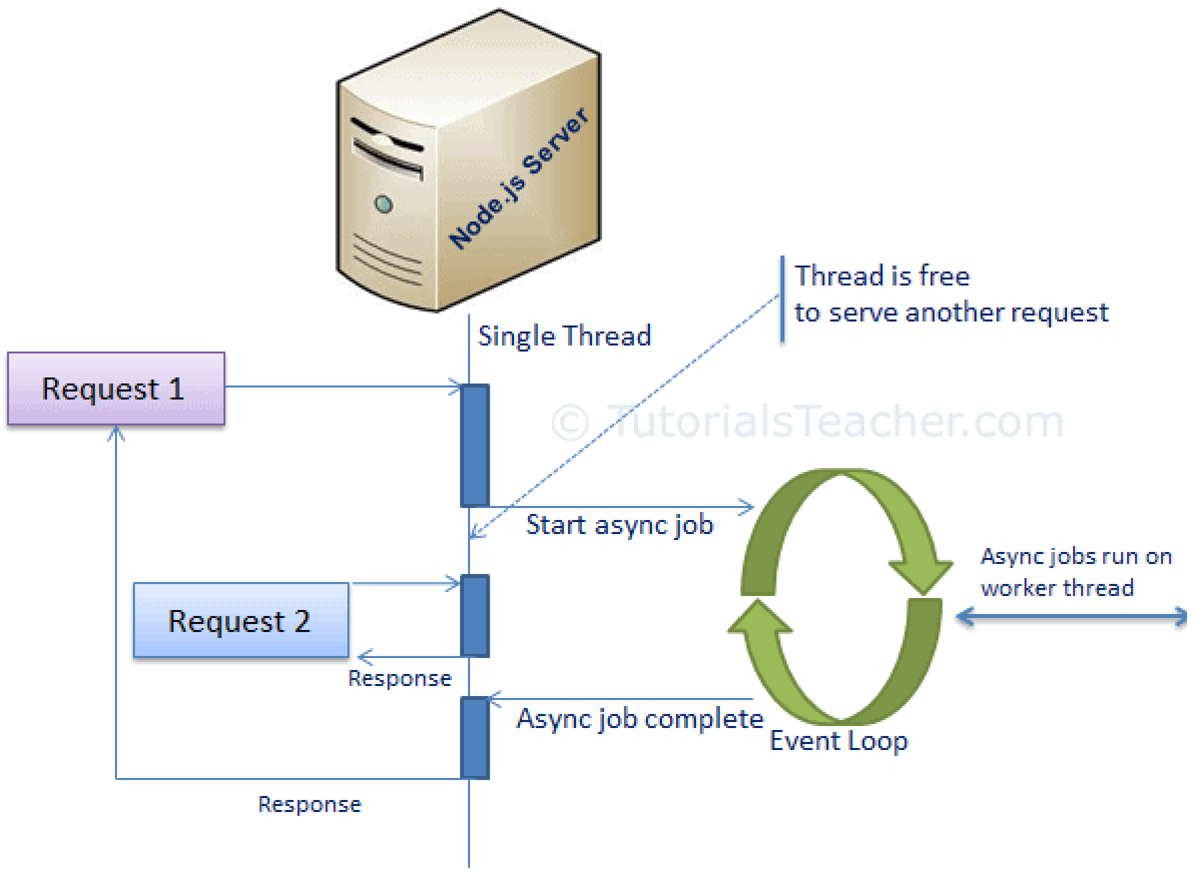
What is unique about Node.js?

- Node.js operates on a single-threaded event loop architecture. This means it uses one main thread to handle all incoming requests instead of creating a new thread for each connection.
- The event loop continuously checks for tasks and handles them one at a time, but it does so very efficiently by managing tasks asynchronously.
- Node.js uses non-blocking I/O operations, which means that when a task (like reading a file or querying a database) is initiated, the event loop does not wait for it to complete. Instead, it registers a callback function and moves on to handle other tasks.
- Once the I/O operation is complete, the callback function is invoked, allowing the server to process the result without blocking the main thread.

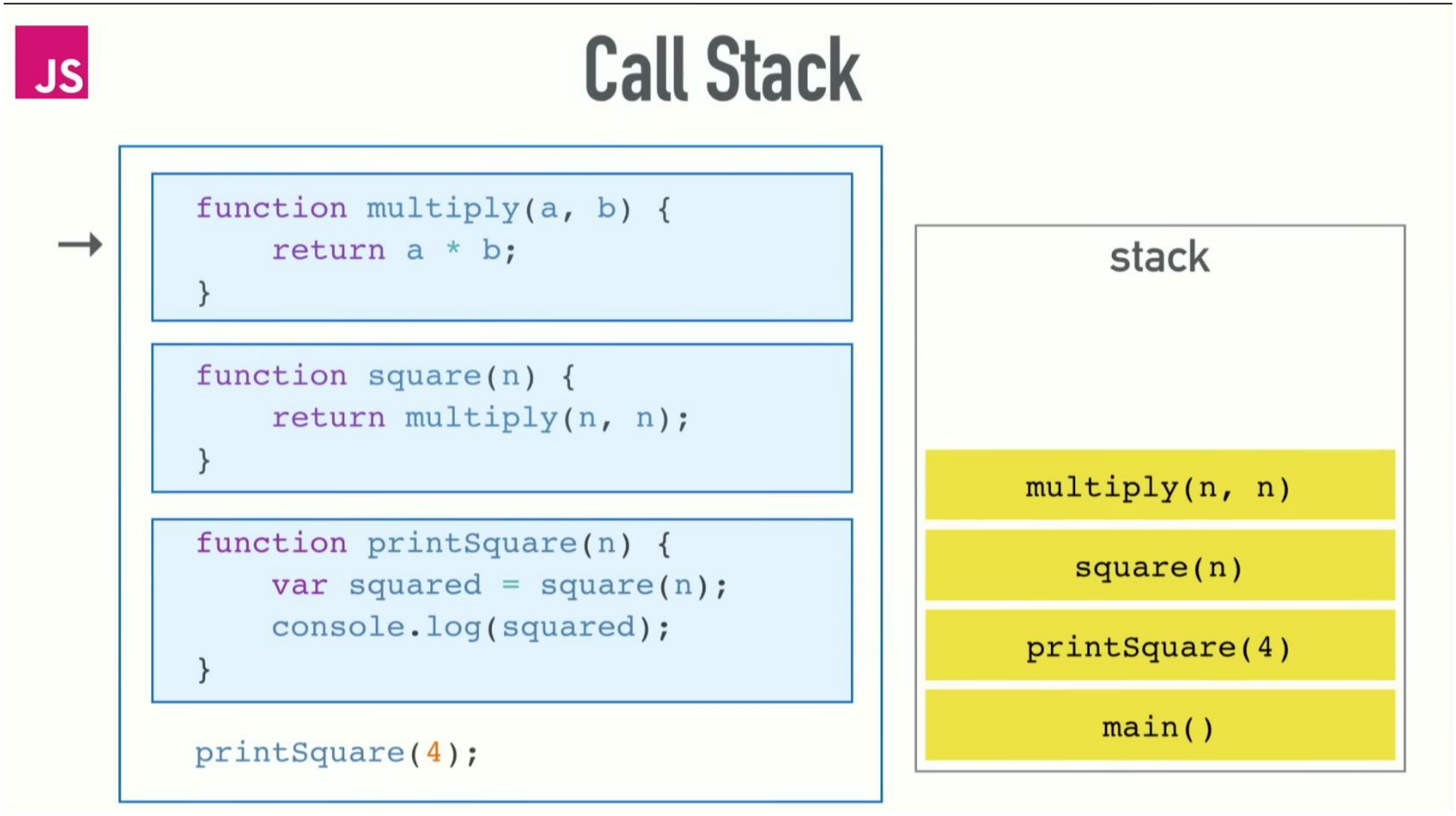
What is unique about Node.js?

- JavaScript on server-side thus making communication between client and server happen in same language
- Servers normally thread based but Node.JS is “Event” based. Node.JS serves each request in a Evented loop that can handle simultaneous requests.





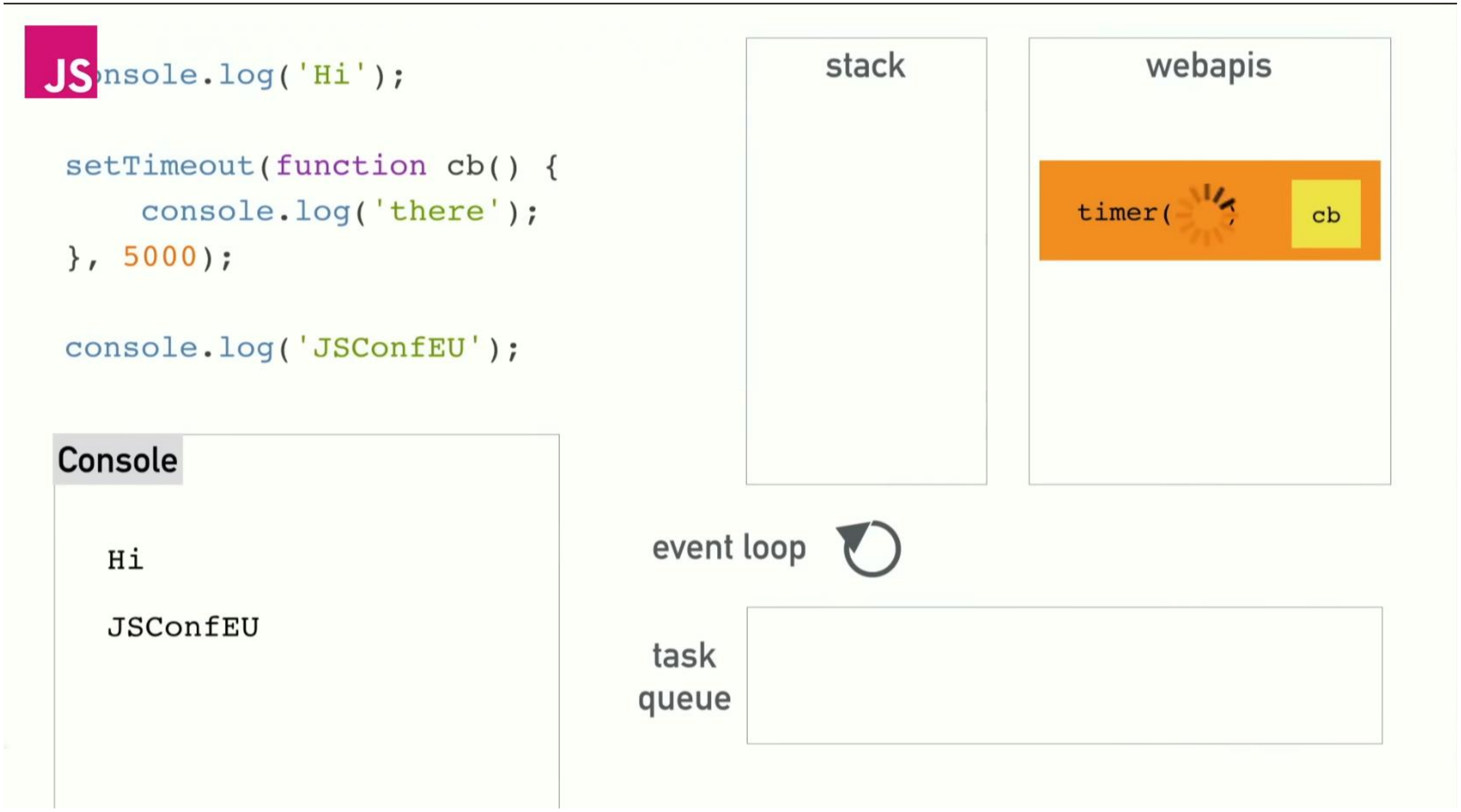
Source: https://www.tutorialsteacher.com/nodejs/nodejs-process-model



Call Stack

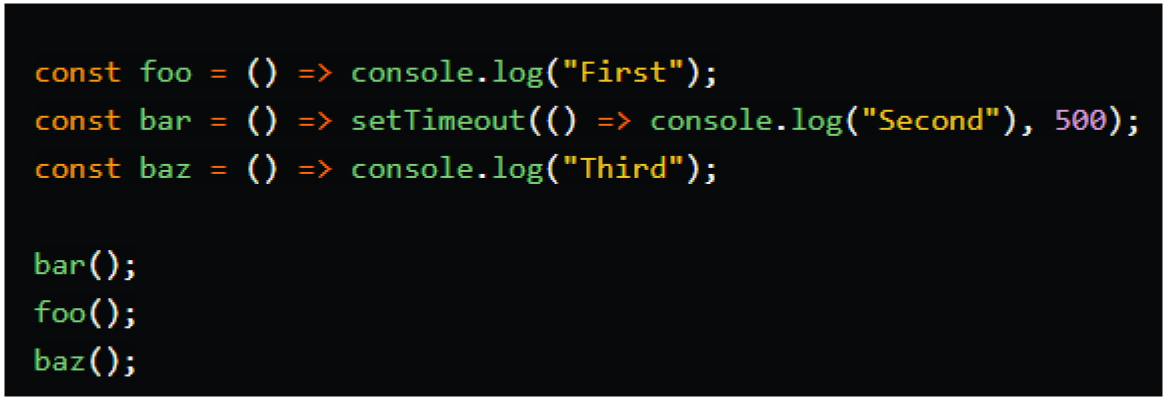
```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```

Source: https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html

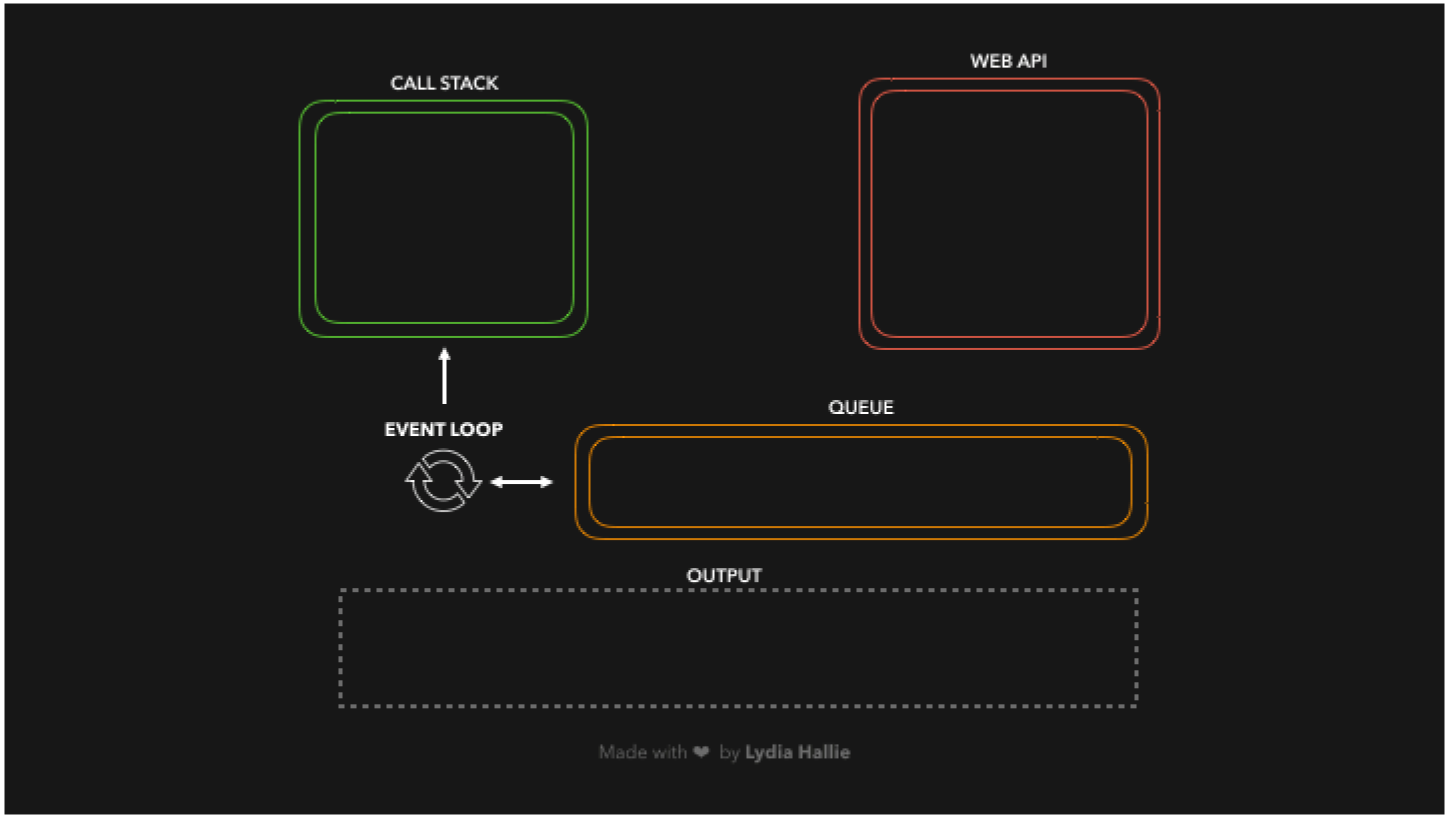


```
console.log('Hi');

setTimeout(function cb() {
    console.log('there');
}, 5000);

console.log('JSConfEU');
```

```
Hi

JSConfEU
```



```
const foo = () => console.log("First");
const bar = () => setTimeout(() => console.log("Second"), 500);
const baz = () => console.log("Third");

bar();
foo();
baz();
```



Source: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif

Why Use Node.js ?

- Node's goal is to provide an easy way to build scalable network programs.
- It lets you layered on top of the TCP library as an HTTP and HTTPS client/server.
- The JS executed by the V8 JavaScript engine (the thing that makes Google Chrome so fast)
- Node provides a JavaScript API to access the network and file system.

What Node can’t do?

- Node is a platform for writing JavaScript applications outside web browsers. This is not the JavaScript we are familiar with in web browsers.
- There is no DOM built into Node, nor any other browser capability.
- Node can’t run on GUI, but run on terminal
- In the Node.js module system, each file is treated as a separate module.

Installing and using node Module

- Before you can start coding with Node.js, you need to install it on your computer. Visit the official Node.js website (https://nodejs.org/) to download the installer for your operating system.
- After the installation is complete, open your terminal or command prompt and verify that Node.js and npm (Node Package Manager) are installed by running the following commands:
  - node -v
  - npm -v

Installing and using node Module

- Install a module…..inside your project directory
  - npm init -y
  - npm install <module name>
- Using module….. Inside your JavaScript code
  - var http = require(‘http’);
  - var fs = require(‘fs’);
  - var express = require(‘express’);

Creating Your First Node.js Application

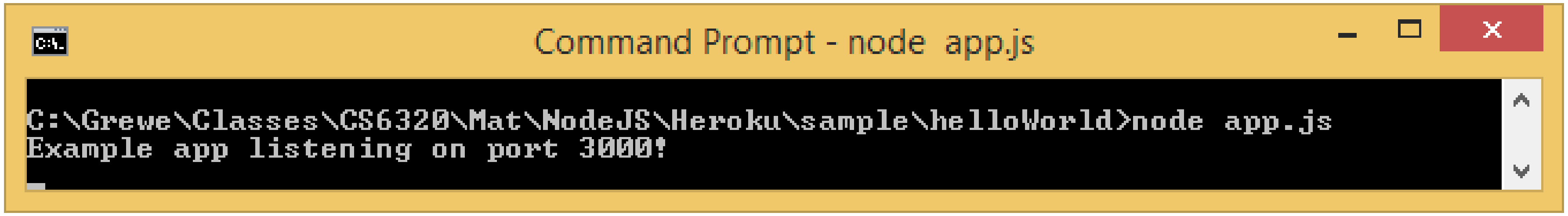
- Create a new directory for your project and navigate to it using the terminal.
  - mkdir my-node-app
  - cd my-node-app
  - npm init –y (*initializes a new Node.js project and automatically creates a default package.json file without asking any questions*)
- Inside your project directory, create a new JavaScript file (e.g., app.js) using your favorite code editor.

Hello World example (app.js)

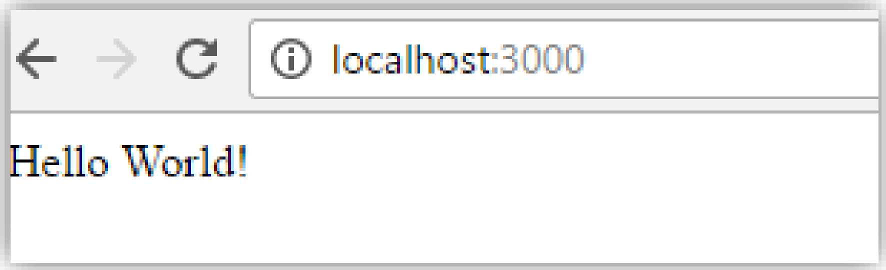
```
const http = require('http’);
const hostname = '127.0.0.1’;
const port = 3000;

const server = http.createServer((req, res) => {
       res.statusCode = 200;
       res.setHeader('Content-Type', 'text/plain’);
       res.end('Hello World’);
});



server.listen(port, hostname, () => {
       console.log(`Server running at http://${hostname}:${port}/`);
});
```

Run your hello world application

- Run the app with the following command:
  - node app.js
- Then, load http://localhost:3000/ in a browser to see the output.


Command Prompt - node app.js

```
C:\Grewe\Classes\CS6320\Mat\NodeJS\Heroku\sample\helloWorld>node app.js
Example app listening on port 3000!
```


localhost:3000

Hello World!

Callback Hell

Callback hell refers to a situation in programming where multiple nested callbacks lead to code that is difficult to read, maintain, and debug.

```
asyncOperation1(function(result1) {
        asyncOperation2(result1, function(result2) {
                asyncOperation3(result2, function(result3) {
                                asyncOperation4(result3, function(result4) {
                                // ... more nested callbacks
                        });
                });
        });
});
```

Callback Hell Example

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

Promise

"I Promise a Result!"

- "Producing code" is code that can take some time
- "Consuming code" is code that must wait for the result
- A Promise is an Object that links Producing code and Consuming code

myPromise.state	**myPromise.result**
"pending"	undefined
"fulfilled"	a result value
"rejected"	an error object

Promise

```
let myPromise = new Promise(function(myResolve,
myReject) {
// "Producing Code" (May take some time)

  if (success) {myResolve(); }// when successful
  else         {myReject(); } // when error
});

// "Consuming Code" (Must wait for a fulfilled
Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Promise

```
let myPromise = new Promise((myResolve,myReject) =>
{
// "Producing Code" (May take some time)

  if (success) {myResolve(); }// when successful
  else         {myReject(); } // when error
});

// "Consuming Code" (Must wait for a fulfilled
Promise)
myPromise.then(
  (value) =>  { /* code if successful */ },
  (error) =>  { /* code if some error */ }
);
```

Promise

```
let myPromise = new Promise((myResolve,myReject) =>
{
// "Producing Code" (May take some time)

  if (success) {myResolve(); }// when successful
  else         {myReject(); } // when error
});

// "Consuming Code" (Must wait for a fulfilled
Promise)
myPromise.then(
  (value) =>  { /* code if successful */ })
  .catch((message) => {/* code if failed */ });
```

Promise

- A Promise is a JavaScript object
  - `executor`: a function that may take some time to complete. After it's finished, it sets the values of `state` and `result` based on whether the operation is successful
  - `state`: "pending" ➔ "fulfilled"/"rejected"
  - `result`: undefined ➔ value/error

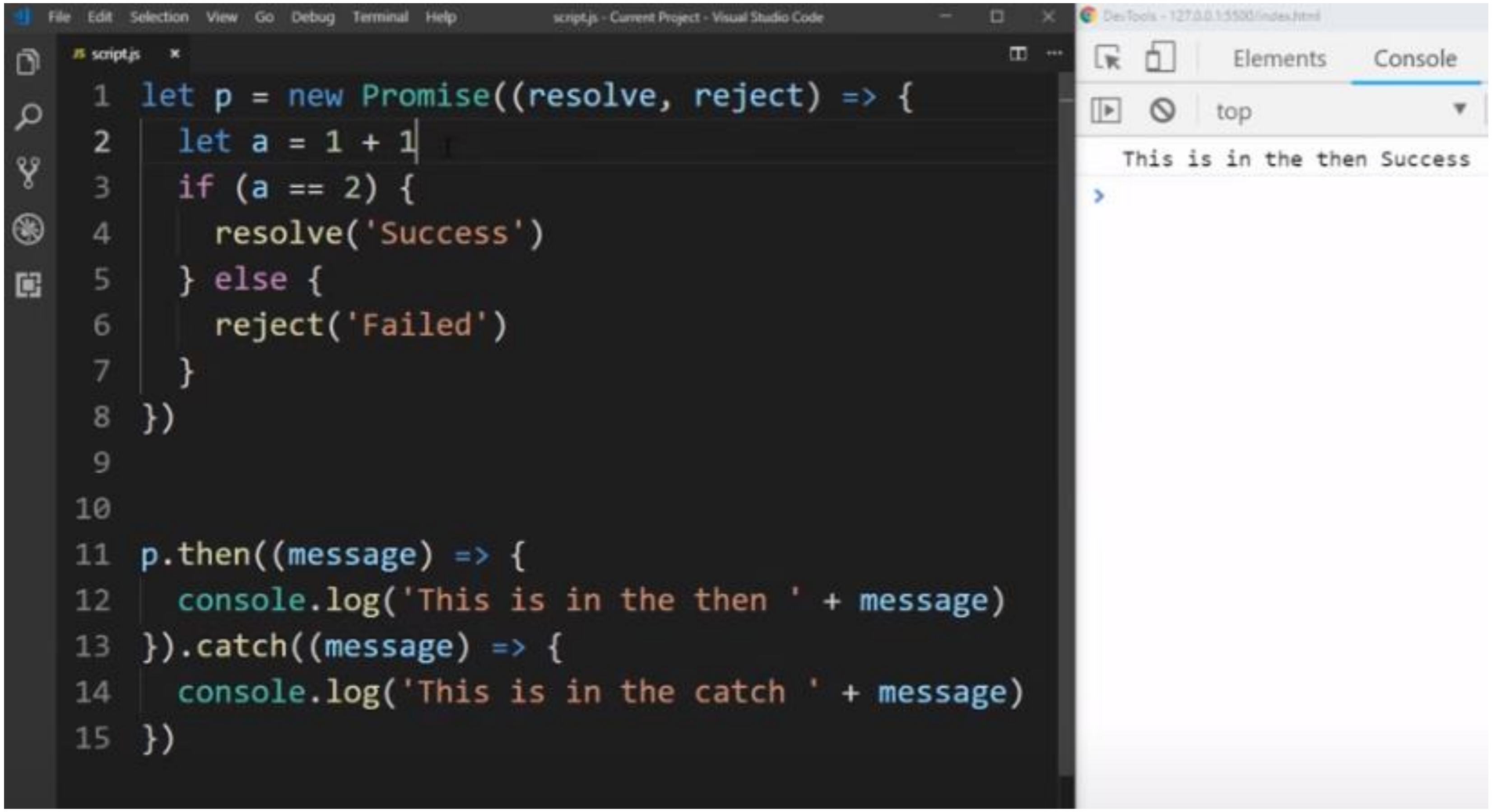
```
var promise = doSomethingAync()
promise.then(onFulfilled, onRejected)
```

Use A Promise

```
promise.then(
  function(result) {/* handle result */},
  function(err) {/* handle error */ }
);
```

- After `executor` finishes, either the success handler or the error handler will be called and `result` will be passed as the argument to the handler.
- With promises, you can chain .then() methods to handle the resolved value and .catch() to handle errors, reducing the nested structure seen in callback hell.

Promise


File Edit Selection View Go Debug Terminal Help
script.js - Current Project - Visual Studio Code
script.js
Elements
Console
top

```
let p = new Promise((resolve, reject) => {
  let a = 1 + 1
  if (a == 2) {
    resolve('Success')
  } else {
    reject('Failed')
  }
})


p.then((message) => {
  console.log('This is in the then ' + message)
}).catch((message) => {
  console.log('This is in the catch ' + message)
})
```

```
This is in the then Success
```

National University
Of Computer and Emerging Sciences

About Promise

- There can only be one result or an error
- Once a promise is settled, the result (or error) never changes
- `then()` can be called multiple times to register multiple handlers

Other Common Usage of Promise

```
promise.then( success_handler );

promise.then( null, error_handler );

promise.catch( error_handler );

promise
  .then( success_handler )
  .catch( error_handler )
```

Callback vs Promise

```
function readFile(callback) {
   const data = "File content read successfully!";
   callback(data); }
function processData(fileContent, callback) {
   const processedData = fileContent.toUpperCase();
     callback(processedData); }
function saveData(processedContent) {
   console.log("Data saved: " + processedContent); }

readFile(function(result) {
   processData(result, function(processedResult) {
      saveData(processedResult);
   }); });
```

Callback vs Promise

```
function readFile() {
   return new Promise((resolve) => {
      const data = "File content read successfully!";
      resolve(data);
   }); }
function processData(fileContent) {
   return new Promise((resolve) => {
      const processedData = fileContent.toUpperCase();
      resolve(processedData);
   }); }
function saveData(processedContent) {
   console.log("Data saved: " + processedContent); }
readFile()
   .then(result => processData(result))
   .then(processedResult => saveData(processedResult));
```

Promises Chaining

- A technique in JavaScript where multiple asynchronous operations are performed one after the other.
- The result of one promise is passed to the next operation, and each subsequent promise depends on the completion of the previous one.
- This avoids the nesting of callbacks (callback hell) and keeps the code linear and readable.
- Suppose we have three functions `f1`, `f2`, `f3`
  - `f1` returns a Promise
  - `f2` relies on the result produced by `f1`
  - `f3` relies on the result produced by `f2`

```
f1.then(f2).then(f3)
```

Understand Promise Chaining …

```
f1.then(f2)
```

- then() returns a Promise based on the return value of the handler function
  - If `f2` return a regular value, the value becomes the result of the Promise
  - If `f2` return a promise, the result of that Promise becomes the result of the Promise returned by `then()`

… Understand Promising Chaining

```
let f1 = new Promise((resolve) => {
    resolve("Step 1 Complete");
});

f1.then((result) => {
    console.log(result);
    return "Step 2 Complete";
})
.then((newResult) => {
    console.log(newResult);
});
```

… Understand Promising Chaining

```
let f1 = new Promise((resolve) => {
    resolve("Step 1 Complete");
});

f1.then((result) => {
    console.log(result);
    return new Promise((resolve) => {
      resolve("Step 2 Complete after
delay"); });
})
.then((newResult) => {
    console.log(newResult);
});
```

National University
Of Computer and Emerging Sciences

Promise with async – await

"async and await make promises easier to write"

async makes a function return a Promise
await makes a function wait for a Promise

await keyword: Pauses the execution of an async function until the promise resolves. It makes the code look as if it's synchronous while still being non-blocking.
async function: A function declared with async automatically returns a promise. Inside an async function, you can use the await keyword.
With async/await, you can write asynchronous code that looks synchronous, making it more readable and easier to understand.

Promise with async – await

```
async function myDisplay() {
 let promise = new Promise(function(resolve, reject) {
 const success = false; // Simulate an error
      if (success) {
          resolve("Data fetched!");
      } else {
          reject("Failed to fetch data!");
      }  });
 try {
      const result = await promise;
      console.log(result); // Logs if successful
   } catch (error) {
      console.error(error); // Logs: "Failed to fetch data!"
   }
}
myDisplay();
```

Promise with async – await

```
async function fetchData() {
    return new Promise((resolve, reject) => {
        const success = false; // Simulate an error
        if (success) {
            resolve("Data fetched!");
        } else {
            reject("Failed to fetch data!");
        } }); }
async function displayData() {
    try {
        const result = await fetchData(); // Wait for the data
        console.log(result); // Logs if successful
    } catch (error) {
        console.error(error); // Logs: "Failed to fetch data!"
    }
}
displayData();
```

.then() vs async – await

```
f1()
    .then(rF1 => {
        return f2(rF1);
    })
    .then(rF2 => {
        return f3(rF2);
    })
    .then(rF3 => {
        console.log(rF3);});
```

```
async function runAsyncTasks() {
    try {
        const rF1 = await f1();
        const rF2 = await f2(rF1);
        const rF3 = await f3(rF2);
        console.log(rF3); }
    catch (error) {
        console.error("Error: ", error);
} }
```

Promise Example 2

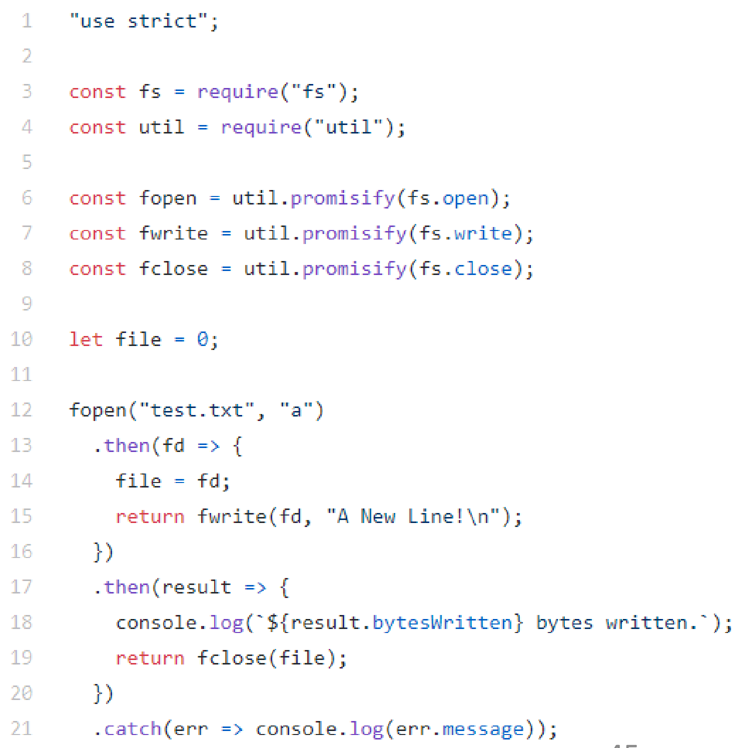
- Promisifying a function in Node.js, also known as "promisification", involves converting a callback-based function into a function that returns a Promise.
  - Use of promisify() in the Utilities package

Original function:

`func(args…, callback(err,result))`

Promisified:

`func(args…) returns a Promise`



```javascript
"use strict";

const fs = require("fs");
const util = require("util");

const fopen = util.promisify(fs.open);
const fwrite = util.promisify(fs.write);
const fclose = util.promisify(fs.close);

let file = 0;

fopen("test.txt", "a")
  .then(fd => {
    file = fd;
    return fwrite(fd, "A New Line!\n");
  })
  .then(result => {
    console.log(`${result.bytesWritten} bytes written.`);
    return fclose(file);
  })
  .catch(err => console.log(err.message));
```

Promise Example 2

```
function add(a, b, callback) {
   setTimeout(() => callback(null, a + b), 100); }

add(1, 2, (err, first) => {
   console.log(first);
   add(first, 3, (err, second) => {
      console.log(second);
      add(second, 4, (err, finalResult) => {
      console.log(finalResult);
      });
   });
});
```

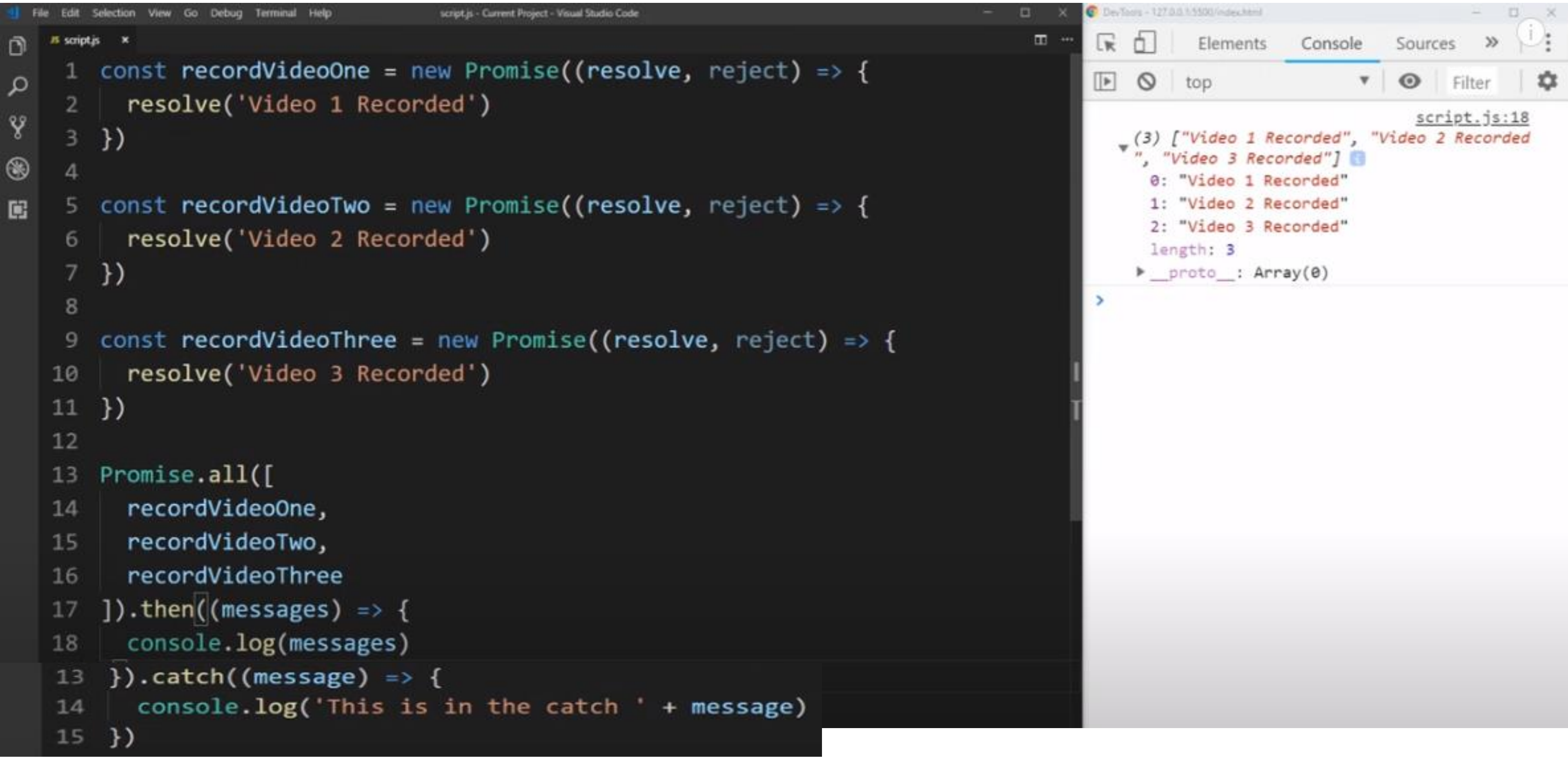
Promise Example 2

```
function add(a, b, callback) {
    setTimeout(() => callback(null, a + b), 100); }

const addPromise = promisify(add);

addPromise(1, 2)
    .then((first) => {
        console.log(first);
        return addPromise(first, 3);  })
    .then((second) => {
        console.log(second);
        return addPromise(second, 4);  })
    .then((finalResult) => {
        console.log(finalResult);})
    .catch((error) => {    t
        console.error('Error:', error);
});
```

Promise Example 2

```
function add(a, b, callback) {
   setTimeout(() => callback(null, a + b), 100); }

const addPromise = promisify(add);
const run = async () => {
   const first = await addPromise(1, 2);
   console.log(first);
   const second = await addPromise(first, 3);
   console.log(second);
   const finalResult = await addPromise(second, 4);
   console.log(finalResult);
};
```

Promise



```
const recordVideoOne = new Promise((resolve, reject) => {
  resolve('Video 1 Recorded')
})

const recordVideoTwo = new Promise((resolve, reject) => {
  resolve('Video 2 Recorded')
})

const recordVideoThree = new Promise((resolve, reject) => {
  resolve('Video 3 Recorded')
})

Promise.all([
  recordVideoOne,
  recordVideoTwo,
  recordVideoThree
]).then((messages) => {
  console.log(messages)
}).catch((message) => {
  console.log('This is in the catch ' + message)
})
```

```
                                                   script.js:18
▼ (3) ["Video 1 Recorded", "Video 2 Recorded
  ", "Video 3 Recorded"]
    0: "Video 1 Recorded"
    1: "Video 2 Recorded"
    2: "Video 3 Recorded"
    length: 3
  ▶ __proto__: Array(0)
>
```

Promise

```
const recordVideoOne = new Promise((resolve, reject) => {
    setTimeout(() => resolve('Video 1 Recorded'), 1000);
});
const recordVideoTwo = new Promise((resolve, reject) => {
    setTimeout(() => resolve('Video 2 Recorded'), 2000);
});
const recordVideoThree = new Promise((resolve, reject) => {
    setTimeout(() => resolve(' Video 3  Recorded'), 1500);
});

Promise.all([ recordVideoOne, recordVideoTwo, recordVideoThree])
.then((messages) => {
    console.log('All videos recorded:', messages); })
.catch((error) => {
    console.error('One of the promises rejected:', error); });
```

Promise

```
const recordVideoOne = new Promise((resolve, reject) => {
    setTimeout(() => resolve('Video 1 Recorded'), 1000);
});
const recordVideoTwo = new Promise((resolve, reject) => {
    setTimeout(() => resolve('Video 2 Recorded'), 2000);
});
const recordVideoThree = new Promise((resolve, reject) => {
    setTimeout(() => reject(‘ Error: Video 3 Not Recorded'), 1500);
});

Promise.allSettled([ recordVideoOne, recordVideoTwo, recordVideoThree])
.then((messages) => {
    console.log('All videos recorded:', messages); });
```

Parallel Execution

```
Promise.all([promise1, promise2 …])
.then(((results) => {
console.log(results);
/* Logs:       [
{ status: 'fulfilled', value: 10 },
{ status: 'fulfilled', value: 20 },
{ status: ‘fulfilles’, value: 30}        ]  */  });


Promise.allSettled([promise1, promise2, promise3])
.then((results) => {
console.log(results);
/* Logs:       [
{ status: 'fulfilled', value: 10 },
 { status: 'fulfilled', value: 20 },
{ status: 'rejected', reason: 'Error' }        ]     */   });
```

Promise

```
const promiseA = new Promise((resolve) => {
  setTimeout(() => resolve('Promise A resolved!'), 500);
});

const promiseB = new Promise((resolve) => {
  setTimeout(() => resolve('Promise B resolved!'), 200);
});

Promise.race([promiseA, promiseB])
  .then((result) => {
    console.log('Race winner:', result); // Logs: "Race winner: Promise B
})
  .catch((error) => {
    console.error('Race failed:', error); // This won't run in this case
  });
```

Running Node.js Server Applications

- Run server applications using nodemon during development
  - Automatically restart the application when changes in the project are detected
- Deploy server applications using pm2
  - Run server applications as managed background processes

Running Node.js Server Applications

- Run server applications using nodemon during development
  - Automatically restart the application when changes in the project are detected
- Deploy server applications using pm2
  - Run server applications as managed background processes

- https://www.geeksforgeeks.org/node-js-nodemon-module/
- https://medium.com/@ayushnandanwar003/deploying-node-js-applications-using-pm2-a-detailed-guide-b8b6d55dfc88

Routing in Nodejs

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

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

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

    const 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");
});
```

Routing in Nodejs

```
const http = require('http');
http.createServer(function (req, res) {
res.writeHead(200, { 'Content-Type': 'text/html' });
const url = req.url;
     const method = req.method;

     // Regular expression to match '/contact/:id'
    const contactUrlPattern = /^\/contact\/(\d+)$/;
    const match = url.match(contactUrlPattern);
 if (method === 'GET' && url === '/contact') {
        res.write(' Welcome to Get');
         res.end();
     }    else if (method === 'POST' && url === '/contact') {
        res.write(' Welcome to Post');
        res.end();
     }   else if (method === 'PUT' && match) {
        const id = match[1]; // Extract the ID from the URL
        res.write(`Updated contact with ID: ${id}`);
        res.end();
    }     else if (method === 'DELETE' && match) {
        const id = match[1]; // Extract the ID from the URL
        res.write(`Deleted contact with ID: ${id}`);
        res.end();
    } else {
        res.write('Hello World!');
         res.end();
     }
}).listen(3000, function () {
     // The server object listens on port 3000
```

Express

- Express is a user-friendly framework that simplifies the development process of Node applications.
- It uses JavaScript as a programming language and provides an efficient way to build web applications and APIs.
- With Express, you can easily handle routes, requests, and responses, which makes the process of creating robust and scalable applications much easier.
- Moreover, it is a lightweight and flexible framework that is easy to learn and comes loaded with middleware options. Whether you are a beginner or an experienced developer, Express is a great choice for building your application.

Express gives ease of functionality

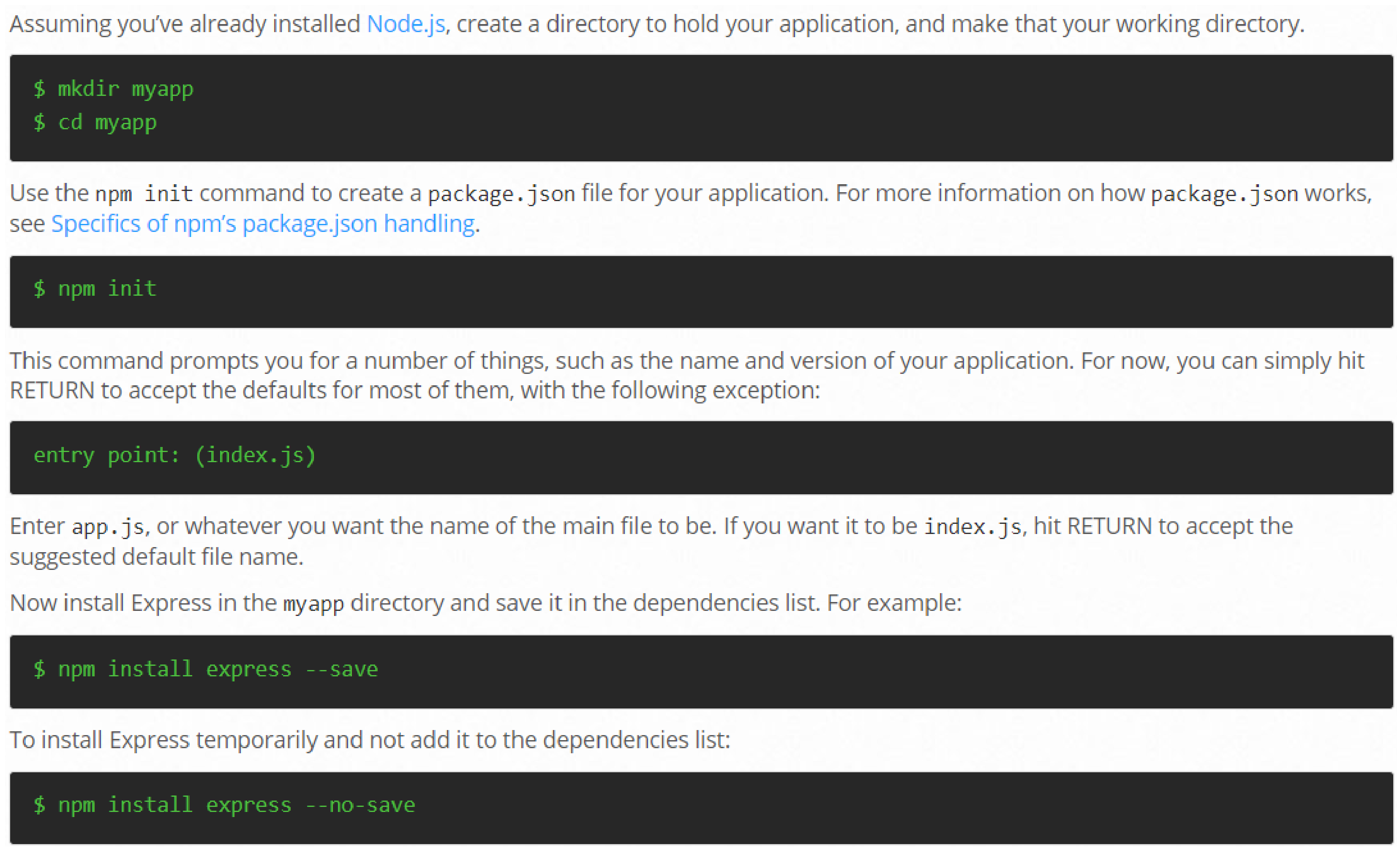
- Routing
- Delivery of Static Files
- “Middleware” – some ease in development (functionality)
- Form Processing
- Simple forms of Authentication
- View support
- Basic error handling, e.g. rejecting malformed requests

A lot of this you can do in NodeJS but, you may write more code to do it than if you use the framework Express.

Express Basics

- Application
- Routing
- Handling requests
- Generating response
- Middleware
- Error handling

Express Installation



Assuming you've already installed Node.js, create a directory to hold your application, and make that your working directory.

```
$ mkdir myapp
$ cd myapp
```

Use the `npm init` command to create a `package.json` file for your application. For more information on how `package.json` works, see Specifics of npm's package.json handling.

```
$ npm init
```

This command prompts you for a number of things, such as the name and version of your application. For now, you can simply hit RETURN to accept the defaults for most of them, with the following exception:

```
entry point: (index.js)
```

Enter `app.js`, or whatever you want the name of the main file to be. If you want it to be `index.js`, hit RETURN to accept the suggested default file name.

Now install Express in the `myapp` directory and save it in the dependencies list. For example:

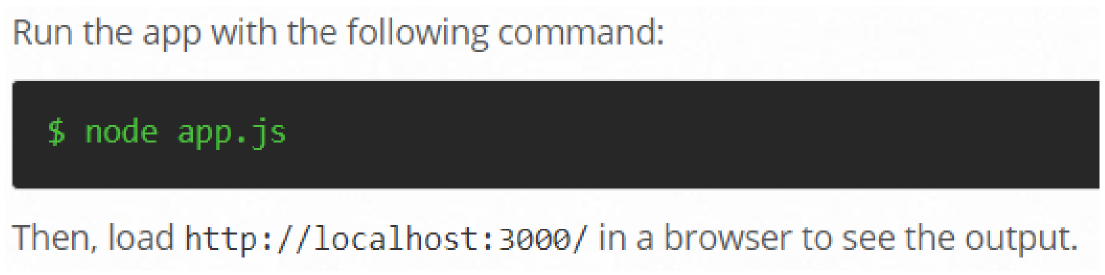
```
$ npm install express --save
```

To install Express temporarily and not add it to the dependencies list:

```
$ npm install express --no-save
```

HelloWorld in Express

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

app.get('/', (req, res) =>
  res.send('Hello World!'));

app.listen(3000, () =>
  console.log('Listening on port 3000'));
```



Run the app with the following command:

```
$ node app.js
```

Then, load `http://localhost:3000/` in a browser to see the output.

Application

```
const app = express();
```

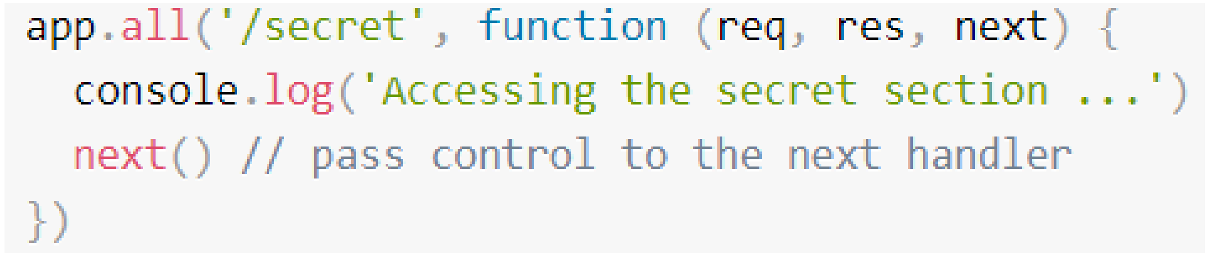
- The Application object
  - Routing requests
  - Static content
  - Configuring middleware

HelloWorld in Express

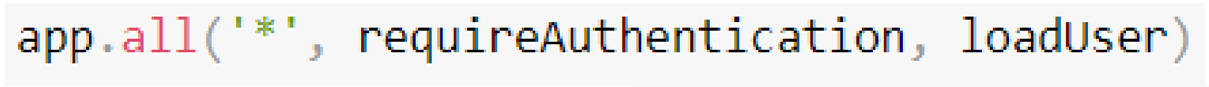
```
const express = require('express');
const app = express();

app.get('/contact', (req, res) =>
  res.send('Hello World!'));
app.post('/contact', (req, res) =>
    res.send('Hello Post!'));
app.put('/contact/:id', (req, res) =>
    res.send('Hello Put!'));
app.delete('/contact/:id', (req, res) =>
    res.send('Hello Delete!'));

app.listen(3000, () =>
  console.log('Listening on port 3000'));
```

Routing Methods in App

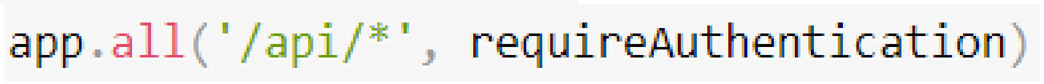
- `app.all( path, callback [,callback …])`



```
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...')
  next() // pass control to the next handler
})
```



```
app.all('*', requireAuthentication, loadUser)
```



```
app.all('/api/*', requireAuthentication)
```

- `app.METHOD( path, callback, [,callback …])`
  - `METHOD` is one of the routing methods, e.g. `get`, `post`, and so on

Modularize Endpoints Using Express Router …

- Example
  - List users: `/users/`, `GET`
  - Add user: `/users/`, `POST`
  - Get user: `/users/:id`, `GET`
  - Delete user: `/users/:id`, `DELETE`

… Modularize Endpoints Using Express Router

- express.Router() is a feature in the Express framework that allows you to create modular route handlers.
- It helps in organizing routes in a cleaner way, especially in larger applications.

```
// routes.js
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
    res.send('List of users’); });
router.post('/', (req, res) => {
    res.send('User created’); });
router.get('/:id', (req, res) => {
    const userId = req.params.id;
    res.send(`User details for user with ID: ${userId}`);
});
router.put('/:id', (req, res) => {
    const userId = req.params.id;
    res.send(`User with ID: ${userId} updated`); });
router.delete('/:id', (req, res) => {
    const userId = req.params.id;
    res.send(`User with ID: ${userId} deleted`); });
module.exports = router;
```

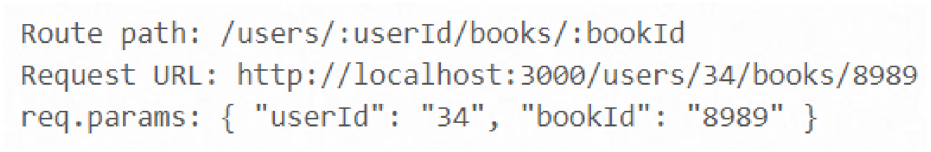
```
// app.js
const express = require('express');
const app = express();
const port = 3000;

// Import contact routes
const routes = require('./routes');

// Use the contact routes
app.use('/contact', routes);

// Start the server
app.listen(port, () => {
    console.log(`Server running at
http://localhost:${port}`);
});
```

Handling Requests

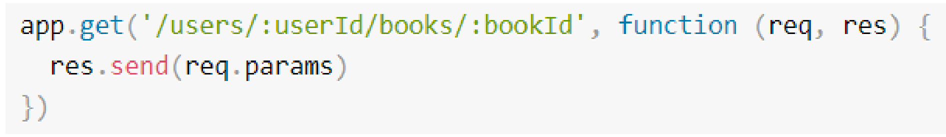
- Request
  - Properties for basic request information such as URL, method, cookies
  - Get header: get()
  - User input
    - Request parameters: req.query
    - Route parameters: req.params
    - Form data: req.body
    - JSON data: req.body

Example: Add

- GET: `/add?a=10&b=20`
- GET: `/add/a/10/b/20`
- POST (Form): `/add`
  - Body: `a=10&b=20`
- POST (JSON): `/add`
  - Content-Type: `application/json`
  - Body: `{"a": 10, "b": 20}`



```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```



```
app.get('/users/:userId/books/:bookId', function (req, res) {
  res.send(req.params)
})
```

```
const express = require('express');
const router = express.Router();

// GET: /add?a=10&b=20
router.get('/add', (req, res) => {
    const { a, b } = req.query;
    res.send(`Sum is: ${+a + +b}`);
});
// GET: /add/a/10/b/20
router.get('/add/a/:a/b/:b', (req, res) => {
    const { a, b } = req.params;
    res.send(`Sum is: ${+a + +b}`);
});
// POST (Form): /add
router.post('/add', express.urlencoded({ extended: true }), (req,
res) => {
    const { a, b } = req.body;
    res.send(`Sum is: ${+a + +b}`);
});
// POST (JSON): /add
router.post('/add', express.json(), (req, res) => {
    const { a, b } = req.body;
    res.send(`Sum is: ${a + b}`);
});
// Export the router
module.exports = router;
```

```
const express = require('express');
const app = express();
const port = 3000;

// Import the add routes
const addRoutes = require('./addRoutes');

// Use the add routes
app.use('/', addRoutes);

// Start the server
app.listen(port, () => {
    console.log(`Server running at 
http://localhost:${port}`);
});
```

Generating Response

- Response
  - Set status: status()
    - end()
  - Send JSON: json()
  - Send other data: send()
  - Redirect: redirect()
  - Other methods for set headers, cookies, download files etc.

```
res.json(null)
res.json({ user: 'tobi' })
res.status(500).json({ error: 'message' })
```

```
res.redirect('/foo/bar')
res.redirect('http://example.com')
res.redirect(301, 'http://example.com')
res.redirect('../login')
```

Generating Response

- Response
  - res.status(404).send('Not Found’);
  - res.end();
  - res.json({ message: 'Hello, world!' });
  - res.send('Hello, world!’);
  - res.redirect('/new-url’);
  - res.setHeader('Custom-Header', 'value’);
  - res.cookie('name', 'value', { httpOnly: true });

Readings

- Express Documentation