Node.js

Introduction

Node.js is a server-side technology that uses JavaScript

Comparison with PHP:

- Node.js application often generates HTML in response to HTTP requests like PHP, but it uses **JavaScript** as its programming language.

- PHP code is typically inserted into HTML markup, which simplifies development. On the other hand, in Node.js, the developer must use `response.write()` to generate HTML.

Two Advantages Over PHP

1) Node.js shines in **push-based web applications**

- In a **pull-based** web application:
  - A web server sits idle until a request is made --> the user pulls information/services from the server
  - The request is the user's job, while the response is the server's job.

- In a **push-based** web application:
  - The server pushes information to the client.
  - Example: chat application.

Example of a Push Web Application

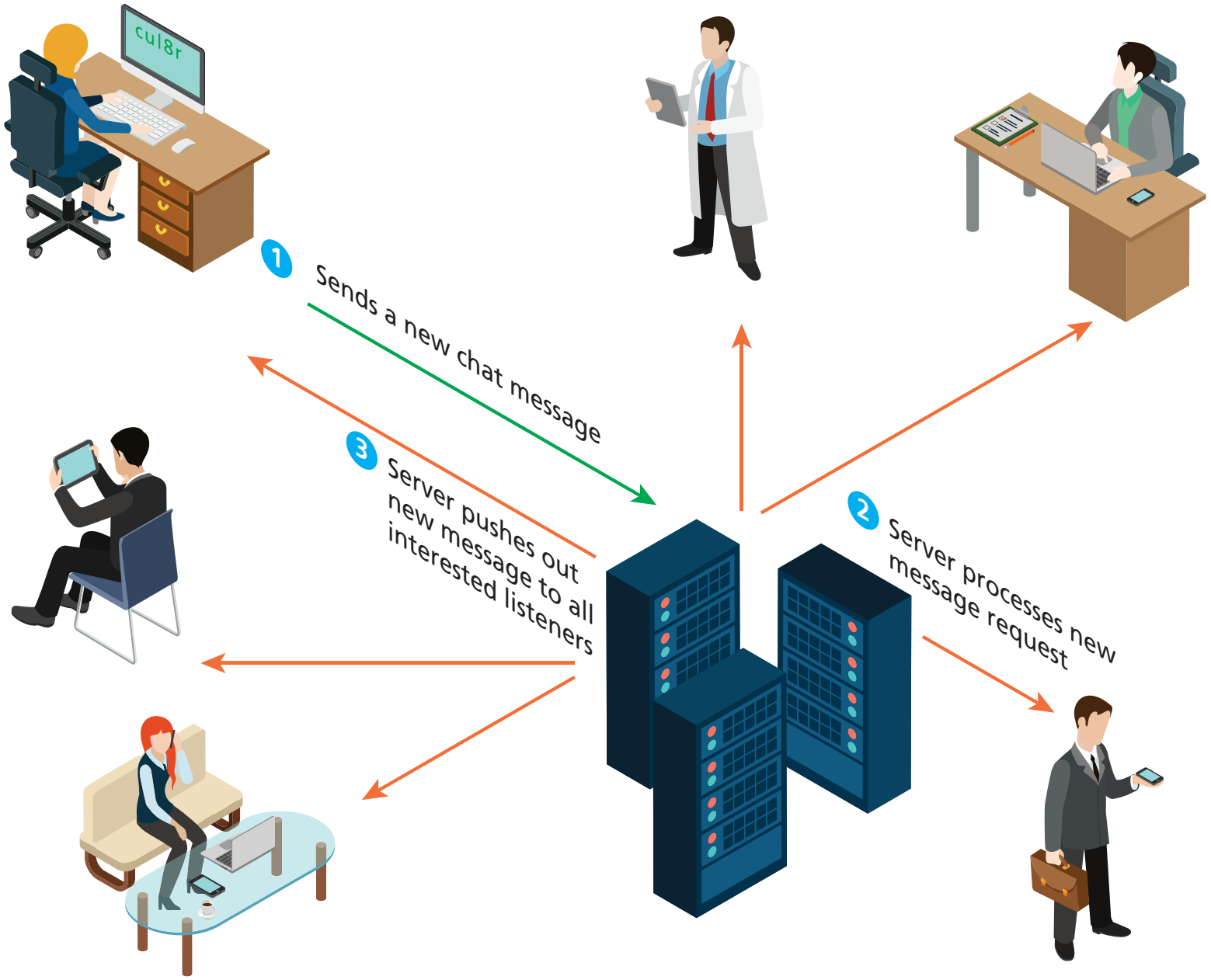


Figure 20.3

Two Advantages Over PHP

2) Node.js uses a **non-blocking, asynchronous, single threaded architecture**.

- Apache runs PHP using either multiprocessing or multithreaded model.
  - Problems:
    - Fixed amount of processes and threads --> a request must wait if none are free.
    - CPU spends a lot of time in context switching in busy sites.
- Node.js uses a single thread
  - Benefit: No time is spent context switching between threads.
- Node.js is a non-blocking asynchronous architecture
  - The tasks are performed asynchronously without blocking

Blocking Thread-based Architecture (PHP)

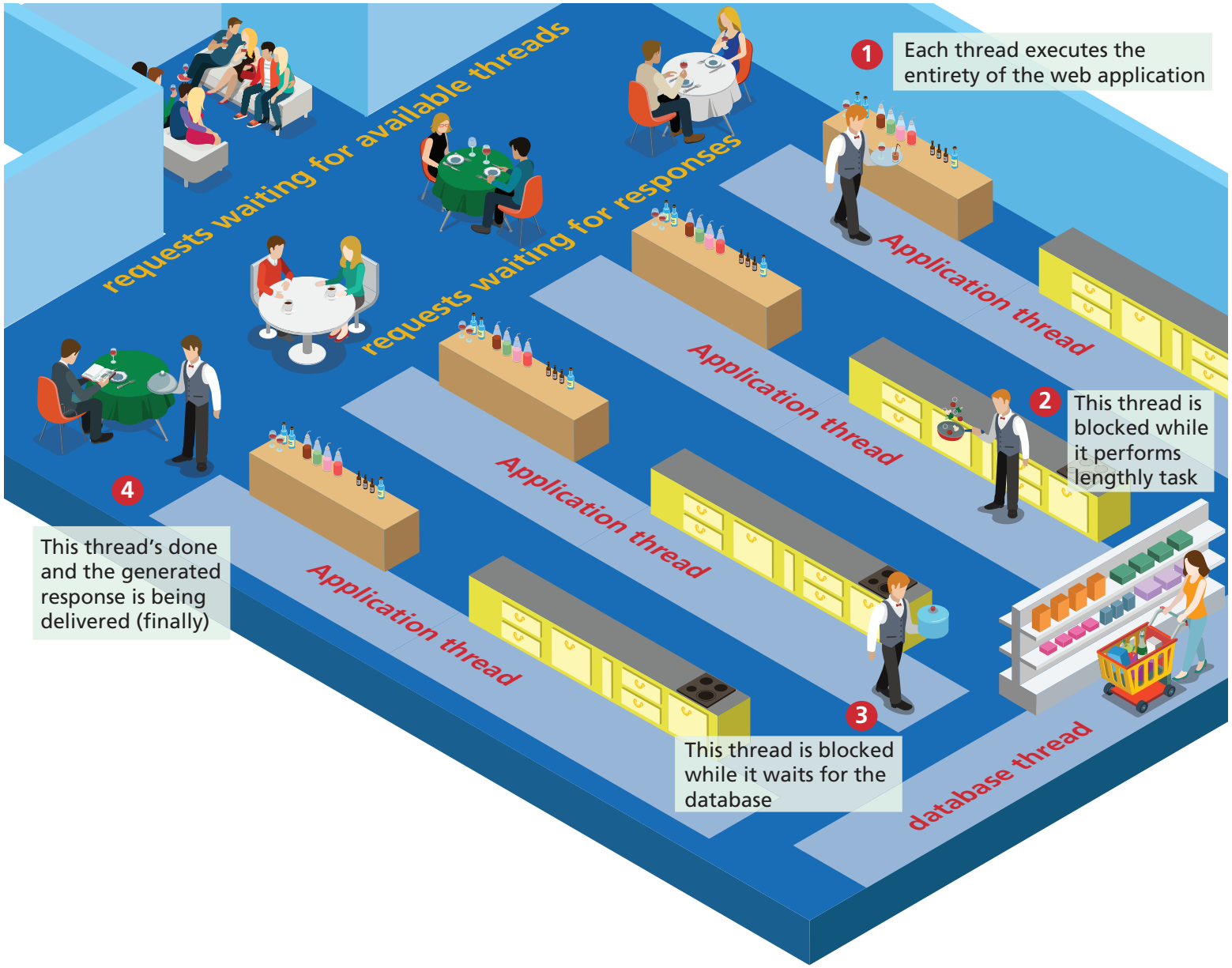


Figure 20.4

Nonblocking Single-Thread Architecture (Node.js)

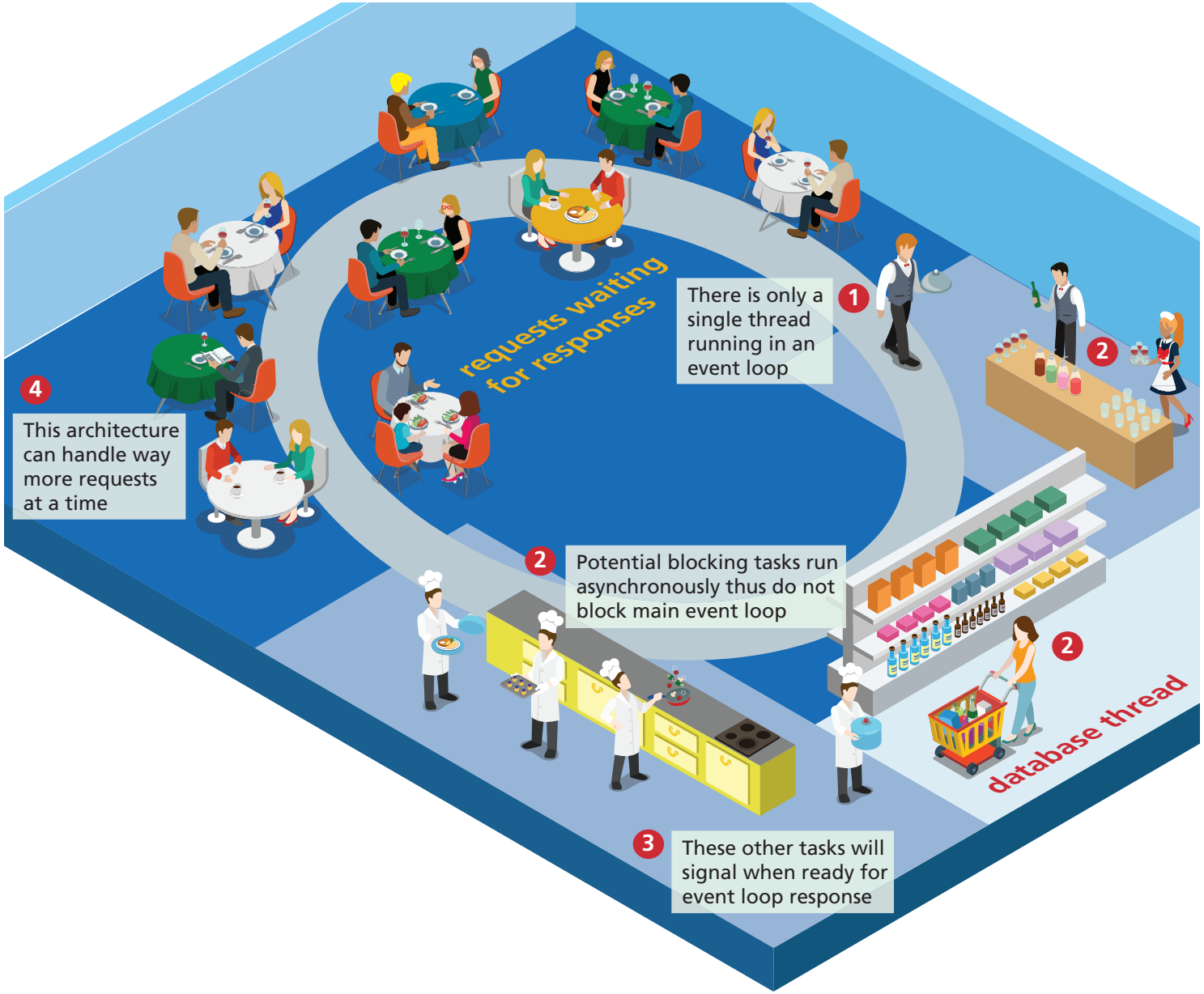


Figure 20.5

In PHP:

```php
if ($result = $db->fetchFromDatabase($sql)){
    // do something with results
    ...
}

if ($data = $service->retrieveFromService($url, $querystring)){
    // do something with data
    ...
}

// doesn't need $result or $data
doSomethingElseReallyImportant();
```

Blocking Calls

Cannot execute until they are finished

In JavaScript:

```
fetchFromDatabase(sql, function(results){
    // do something with results
        ...
});

retrieveFromService(url, querystring, function(data){
    // do something with data
        ...
})

// doesn't need result or data
doSomethingElseReallyImportant();
```

Use callback functions

This isn't blocked by 2 previous calls

Who is using Node.js?

It is not ideal for traditional informational websites.

It is more suited to:

- Data-intensive real-time applications that interact with distributed computers.
- Providing web services.
- Applications that rely on extensive back-end processing.
- Applications that need fast real-time push-based responses, such as mobile games or messaging programs.

Simple Application: Hello World

We will use **Node.js with NetBeans** in this course.

- Installation instructions are provided.

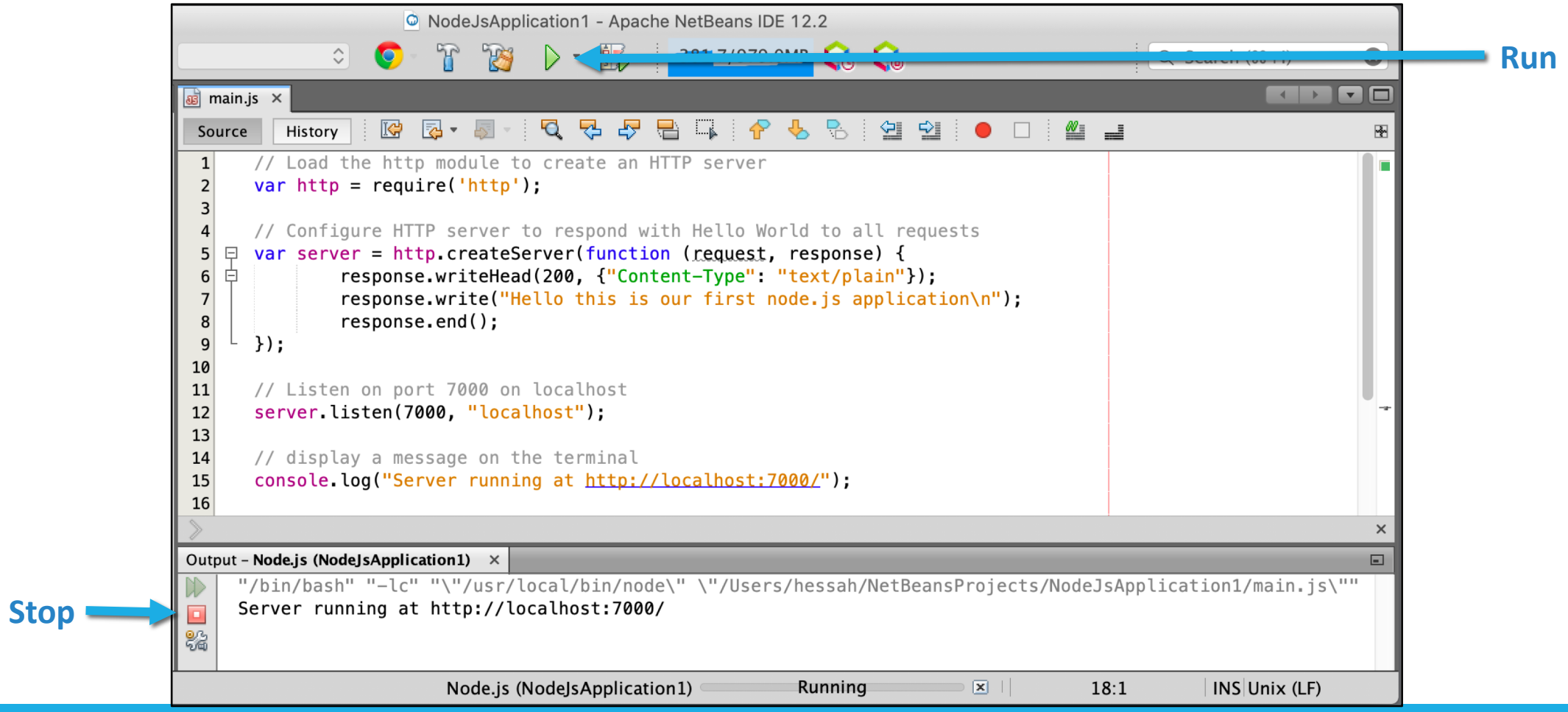
After creating a Node.js project, it will contain the empty file "main.js".

Enter the code shown in the next slide in "main.js".

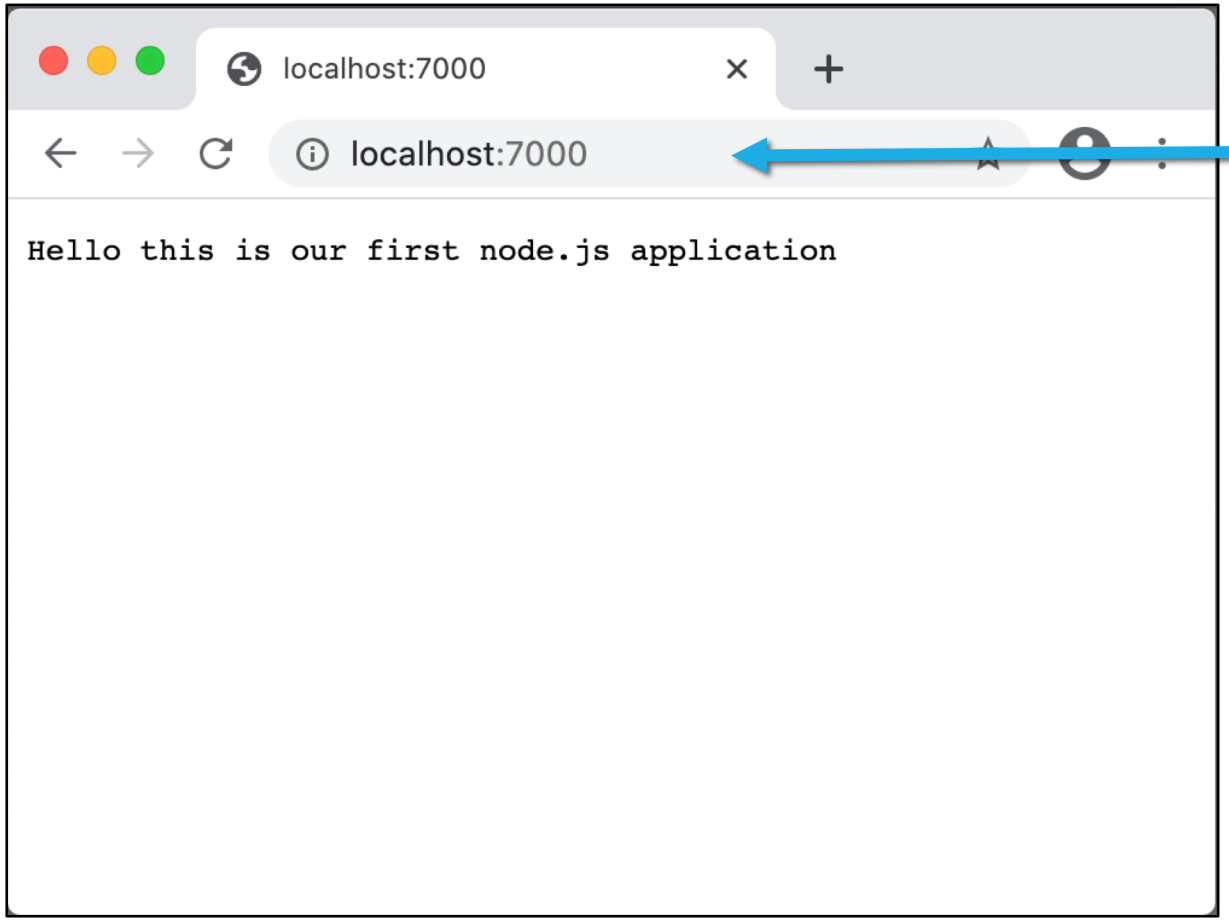
Simple Application: Hello World

```
// Load the http module to create an HTTP server
var http = require('http');

// Configure HTTP server to respond with Hello World to all requests
var server = http.createServer(function (request, response) {
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write("Hello this is our first node.js application\n");
        response.end();
});

// Listen on port 7000 on localhost
server.listen(7000, "localhost");

// display a message on the terminal
console.log("Server running at http://localhost:7000/");
```

Step#1: Run the Program



Step#2: Use Browser to Request URL and Port



http://localhost:7000

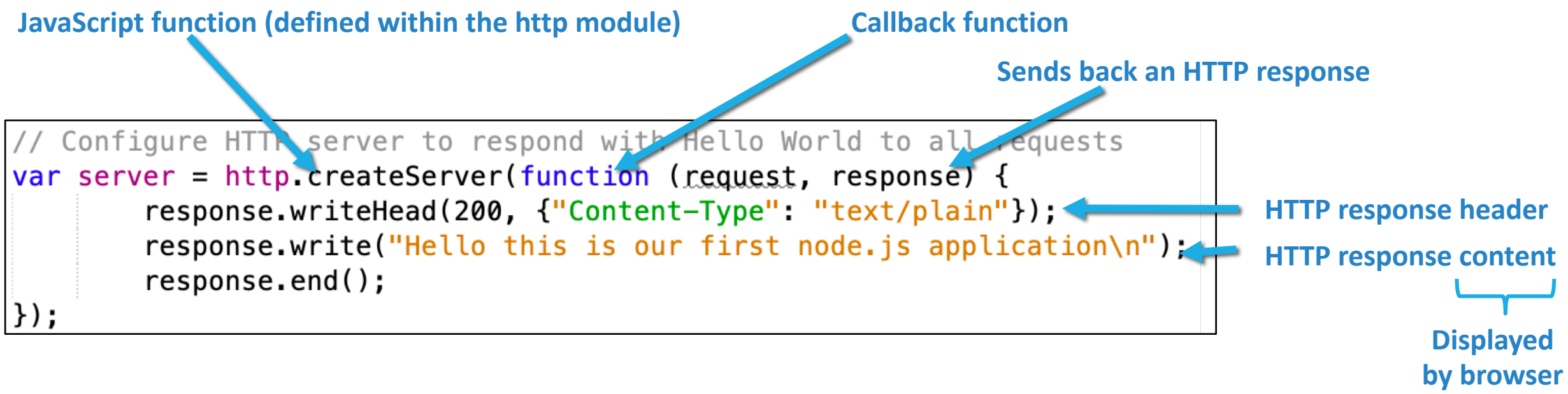
What is the code doing?

```
// Load the http module to create an HTTP server
var http = require('http');
```

`require('http')` --> use a module named `http`

- A module is a JavaScript function library.
- Node.js core includes several important modules.
- Additional modules can be installed, and require the use of npm (Node Package Manager).

What is the code doing?



Node.js as a Web Server

The HTTP module can create an **HTTP server** that **listens to server ports** and gives a **response** back to the client.

Use the `createServer()` method to create an HTTP server

The **function** passed into the `createServer()` method, will be executed when someone tries to access the computer on the specified port (7000).

Add an HTTP Header

You should include an **HTTP header** in the response with the correct content type:

```
response.writeHead(200, {'Content-Type': 'text/html'});
```

Parameters:

- The status code
  - 200 means that all is OK
- An object containing the response headers.

Read the Query String

The **request** parameter in the function passed into the `createServer()`:

- represents the request from the client
- an object of `http.IncomingMessage`

This object has a property called "**url**" which holds the part of the url that comes after the domain name

- Example: If the URL was: http://localhost:8080/summer
  - `request.url` will have the value "/summer"

Split the Query String

The built-in **URL module** can be used to easily split the query string into readable parts.

To include the URL module, use the `require()` method:

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

Parse an address with the `url.parse()` method, and it will return a URL object with each part of the address as properties.

Example

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

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

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

Another Example

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  var q = url.parse(req.url, true).query;
  var txt = q.year + " " + q.month;
  res.end(txt);
}).listen(8080);
```

- If the URL was: http://localhost:8080/?year=2017&month=July
  - The result will be:

```
2017 July
```

Node.js File System Module

The Node.js **file system** module allows you to work with the file system on your computer.

To include the File System module, use the `require()` method:

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

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

Read Files

The `fs.readFile()` method is used to read files on your computer.

Assume that we have the HTML file "demofile1.html" on the server, the following Node.js file reads the HTML file, and returns the content:

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('demofile1.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

Read Files

demofile1.html

```
<html>
<body>
<h1>My Header</h1>
<p>My paragraph.</p>
</body>
</html>
```

Dispayed in Browser:

Serving HTML Files Based on URL of Request – An Example

Assume we have the following two HTML files that are saved in the same folder as the node.js files.

summer.html

```
<!DOCTYPE html>
<html>
<body>
<h1>Summer</h1>
<p>I love the sun!</p>
</body>
</html>
```

winter.html

```
<!DOCTYPE html>
<html>
<body>
<h1>Winter</h1>
<p>I love the snow!</p>
</body>
</html>
```

Serving HTML Files Based on URL of Request – An Example

We can create a Node.js file that opens the requested file and returns the content to the client.

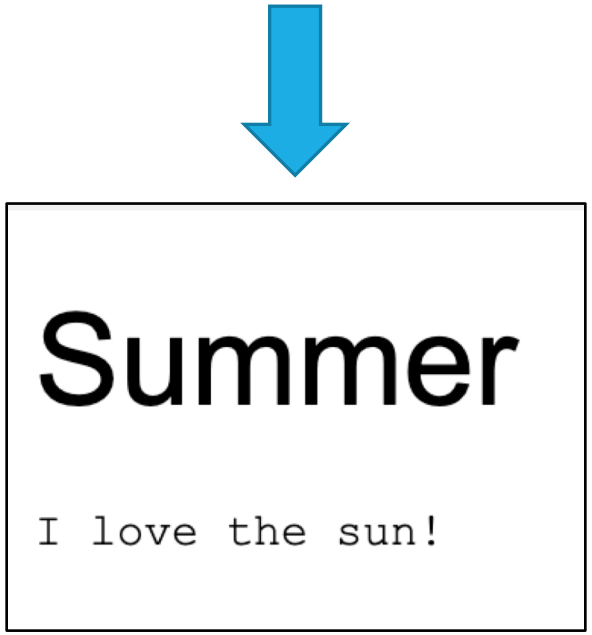
If anything goes wrong, we can throw a 404 error.

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

http.createServer(function (req, res) {
  var q = url.parse(req.url, true);
  var filename = "." + q.pathname;
  fs.readFile(filename, function(err, data) {
    if (err) {
      res.writeHead(404, {'Content-Type': 'text/html'});
      return res.end("404 Not Found");
    }
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

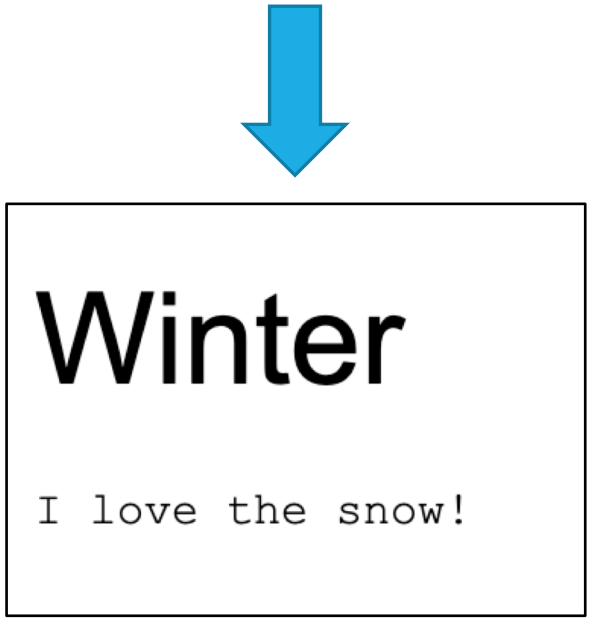
https://www.w3schools.com/nodejs/nodejs_url.asp

Serving HTML Files Based on URL of Request – An Example

http://localhost:8080/summer.html



http://localhost:8080/winter.html



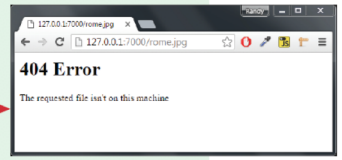
https://www.w3schools.com/nodejs/nodejs_url.asp

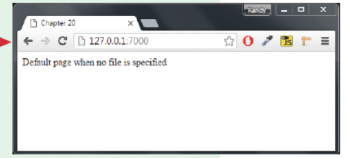
A More Complicated Example

fileserver.js

```
var http = require("http");
var url = require("url");
var path = require("path");
var fs = require("fs");

// our HTTP server now returns requested files
var server = http.createServer(function (request, response) {

    // get the filename from the URL
    var requestedFile = url.parse(request.url).pathname;
    // now turn that into a file system file name by adding the current
    // local folder path in front of the filename
    var filename = path.join(process.cwd(), requestedFile);

    // check if it exists on the computer
    fs.exists(filename, function(exists) {
        // if it doesn't exist, then return a 404 response
        if (! exists) {
            response.writeHead(404,
               {"Content-Type": "text/html"});
            response.write("<h1>404 Error</h1>\n");
            response.write("The requested file isn't on this machine\n");
            response.end();
            return;
        }

        // if no file was specified, then return default page
        if (fs.statSync(filename).isDirectory())
            filename += '/index.html';

        // file was specified then read it in and send its
        // contents to requestor
        fs.readFile(filename, "binary", function(err, file) {
            // maybe something went wrong ...
            if (err) {
                response.writeHead(500, {"Content-Type": "text/html"});
                response.write("<h1>500 Error</h1>\n");
                response.write(err + "\n");
                response.end();
                return;
            }
            // ... everything is fine so return contents of file
            response.writeHead(200);
            response.write(file, "binary");
            response.end();
        });
    });

});
server.listen(7000, "localhost");
console.log("Server running at http://127.0.0.1:7000/");
```

Using two new modules in this example that process URL paths and read/write local files.





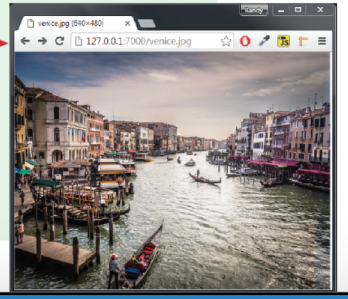


Figure 20.5

Node.js Events

Node.js is perfect for **event-driven applications**.

Node.js has a built-in module, called "**Events**", where you can create, fire, and listen for your own events.

To include the Events module use the `require()` method:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
```

- All event properties and methods are an instance of an EventEmitter object.

The EventEmitter Object

To **assign event handlers** to your own events:

- Use the `on()` method

To **fire an event**:

- Use the `emit()` method.

Example

This example creates a function that will be executed when a "scream" event is fired.

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

//Create an event handler:
var myEventHandler = function () {
  console.log('I hear a scream!');
}

//Assign the event handler to an event:
eventEmitter.on('scream', myEventHandler);

//Fire the 'scream' event:
eventEmitter.emit('scream');
```

Node.js and MySQL

Node.js can be used in **database applications**.

To access a MySQL database with Node.js, you need a MySQL driver.

- You can use the "**mysql**" module, that can be downloaded from NPM.

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

Create Connection

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

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
});
```

Query a Database

```
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("Result: " + result);
  });
});
```

Example: Selecting From a Table

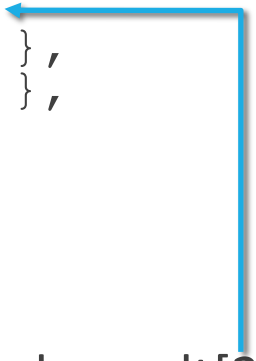
```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword",
  database: "mydb"
});

con.connect(function(err) {
  if (err) throw err;
  con.query("SELECT * FROM customers", function (err, result, fields) {
    if (err) throw err;
    console.log(result);
  });
});
```

https://www.w3schools.com/nodejs/nodejs_mysql_select.asp

The Result Object

The result object is an **array** containing each row as an **object**.

```
[
  { name: 'John', address: 'Highway 71'},
  { name: 'Peter', address: 'Lowstreet 4'},
  { name: 'Amy', address: 'Apple st 652'},
  { name: 'Hannah', address: 'Mountain 21'},
  { name: 'Richard', address: 'Sky st 331'},
  { name: 'Susan', address: 'One way 98'},
  { name: 'Ben', address: 'Park Lane 38'},
]
```



To return the address of the third record: result[2].address

WebSockets with Node.js

One of the key benifts of Node.js --> creating **push-based applications**

Partly reliant on **WebSockets**

- A browser feature supported by all current browsers
- API to open a **two-way** communication channel between the browser and server
- A way for the server to **send/push content** to a client without the client requesting it first

The **Socket.io** module can be used.

Example: Chat Application

Consists of 2 files:

chat-server.js

- The Node.js application that will receive and then push out received messages

chat-client.html

- The server application send it when the browser makes a request
- It contains the user interface that sends and recieves the chat messages

Socket.io contains 2 JavaScript APIs: browser and server.

Chat in the Browser

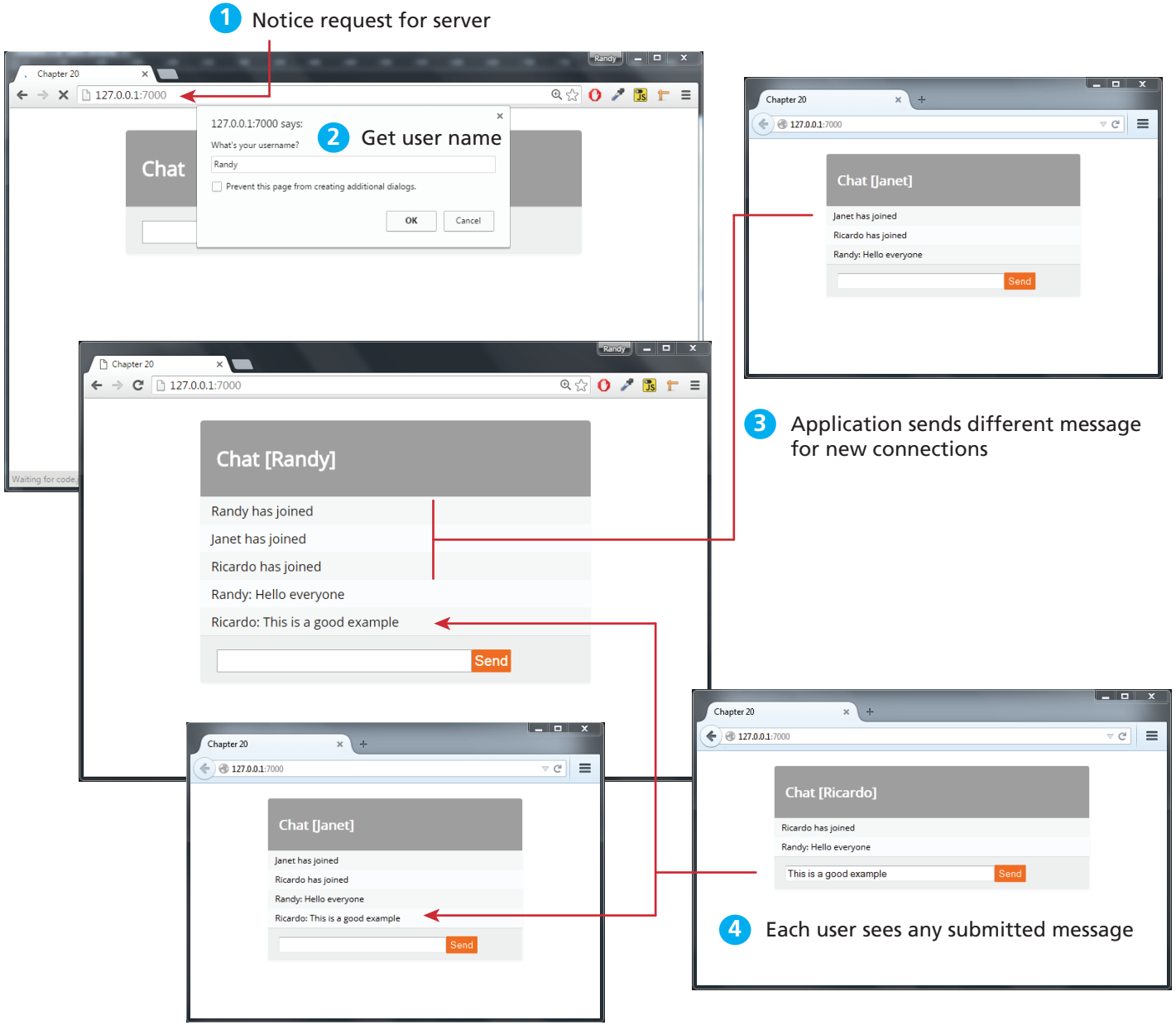


Figure 20.8

chat-server.js

```
var express = require('express');
var app = express();
var http = require('http').Server(app);
var io = require('socket.io')('http');

// tell express where to find static files such as CSS
app.use(express.static('public'));
// every time it receives a request, it sends back the chat client
app.get('/',function(req,res){
    res.sendFile(__dirname + '/chat-client.html');
});
```

Continued in next slide.

chat-server.js

`.on()` method handles the reception of messages

Parameters:

- **Message type**
  - You can specify them (this example has 2 types: username and chat message)
- **Callback function**
  - What to do when the message is received

`emit()`
Broadcast/push a message to all connected clients

```
// handle all WebSocket events, each client is given a unique socket
io.on('connection', function(socket){

    // client has sent username message
    socket.on('username', function(msg){
        // save username for this socket
        socket.username = msg;
        // broadcast message to all connected clients
        io.emit('chat message',msg+ "has joined");
    });

    //client has sent a chat message -> broadcast it
    socket.on('chat message', function(msg){
        io.emit('chat message', socket.username+": "+msg);
    });
});
http.listen(7000, function(){
    console.log('listening on 7000');
});
```

chat-client.html

```
<head>
    ...
    <script src="/socket.io/socket.io.js"></script>

</head>
<body>
    <div class="panel"></div>
        <div class="panel-header"><h3>Chat</h3></div>
        <div class="panel-body"><ul id="messages"></ul></div>
        <div class="panel-footer">
            <form>
                <input type="text" id="entry" />
                <button>Send</button>
            </form>
        </div>
    </div>
```

Area to display received messages

Form to submit messages

Continued in next slide.

chat-client.html

Send messages to the server

Handle messages that have been pushed to the client

```
    <script>
        // initiate WebSocket connection
        var socket = io();

        // get user name and then notify the server
        var username = prompt('What\'s your username?');
        $('.panel-header h3').html('Chat ['+username+']');
        socket.emit('username',username);

        // user has entered a new message
        $('form').submit(function(){
            // send it to the server
            socket.emit('chat message', $('#entry').val());
            //clear text box after submit
            $('#entry').val('');
            // and cancel the submit
            return false;
        });

        // a new chat message has been received
        socket.on('chat message', function(msg){
           $('#messages').append($('<li>').html(msg));
        });
    </script>
</body>
```

References

- "Fundamentals of Web Development" by Randy Connolly and Ricardo Hoar
  - Section 20.2: Node.js

- W3School Node.js Tutorial
  - https://www.w3schools.com/nodejs/default.asp

- Node.js
  - https://nodejs.org/en/