



Full Stack Software Development

Course: Server-Side
Development Using Node.js,
Express.js and MongoDB

Lecture on: Routes Serving
Dynamic Web Applications

Instructor: Rocky Jagtiani



In the previous class...

- You learnt how to route static web applications

Poll 1 (15 Sec)

Modules that do not need to be installed separately and can be used directly are called _____

1. Third-party modules
2. Custom modules
3. Core modules

Poll 1 (Answer)

Modules that do not need to be installed separately and can be used directly are called _____

1. Third-party modules
2. Custom modules
3. **Core modules**

Poll 2 (15 Sec)

Which core module is used to serve static HTML files?

1. http
2. buffer
3. events
4. fs

Poll 2 (Answer)

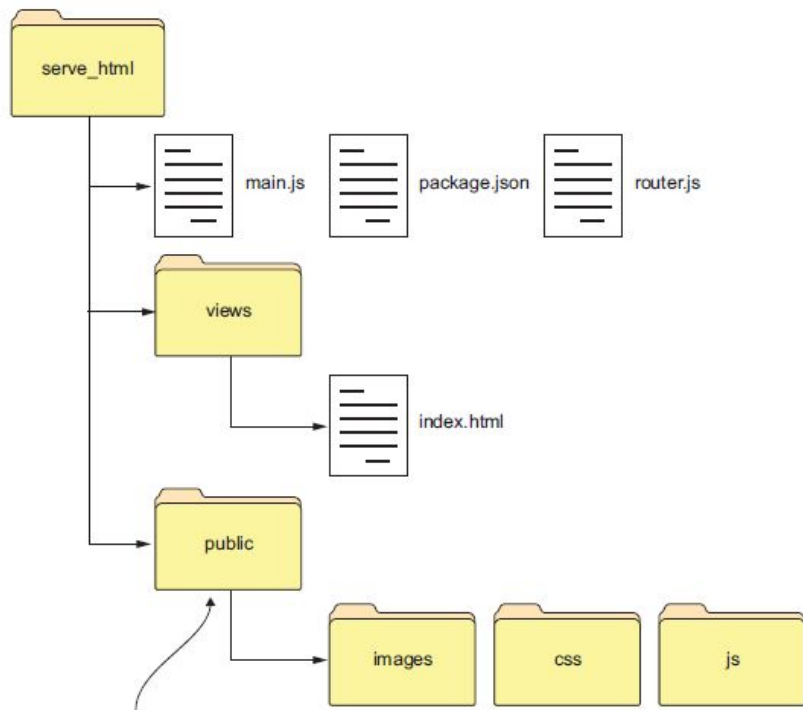
Which core module is used to serve static HTML files?

1. http
2. buffer
3. events
4. **fs**

Today's Agenda

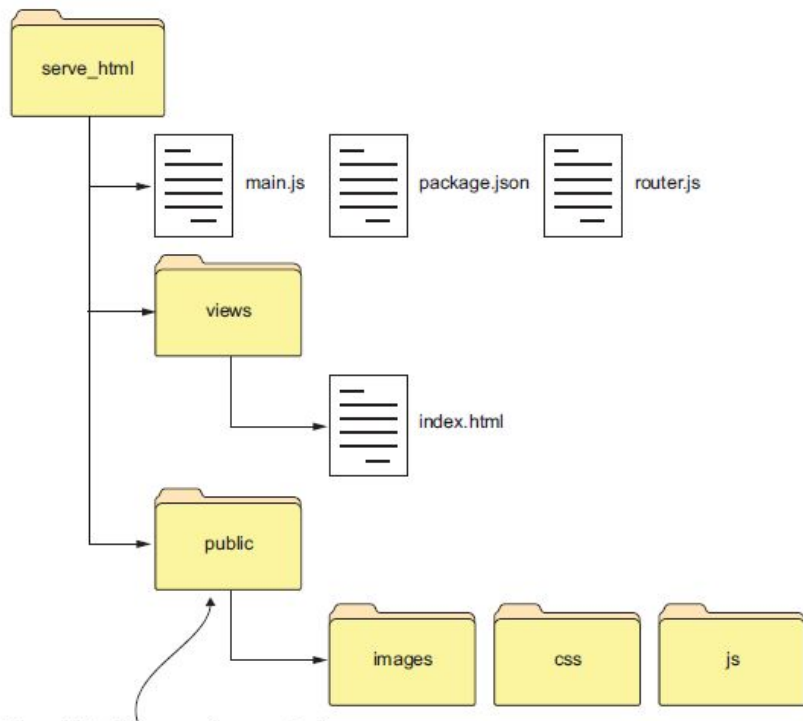
- Routing
- Core modules - events
- EventEmitter class





The public folder can be organised to separate your most common assets served to the client

- The different assets of a web page on the client side, along with its HTML code, include:
 - Images,
 - Stylesheets and
 - Scripts (JavaScript)



The public folder can be organised to separate your most common assets served to the client

- It is necessary that the entire project is arranged in an organised manner for better readability and scalability:
 - We would create a views folder containing all the HTML files
 - We would create a public folder in the project's root directory and move all the assets there
 - Within the public folder, we would create a folder for images, css and js, and move each asset into its respective folder
 - The routing code will form the controller
 - Database connectivity would form the model part of the project

```
const sendErrorResponse = res => {
  res.writeHead(httpStatus.NOT_FOUND, {
    "Content-Type": "text/html"
  });
  res.write("<h1>File Not Found!</h1>");
  res.end();
};

http
  .createServer((req, res) => {
    let url = req.url;
    if (url.indexOf(".html") !== -1) {
      res.writeHead(httpStatus.OK, {
        "Content-Type": "text/html"
      });
      customReadFile(`./views${url}`, res);
    } else if (url.indexOf(".js") !== -1) {
      res.writeHead(httpStatus.OK, {
        "Content-Type": "text/javascript"
      });
      customReadFile(`./public/js${url}`, res);
    } else if (url.indexOf(".css") !== -1) {
      res.writeHead(httpStatus.OK, {
        "Content-Type": "text/css"
      });
      customReadFile(`./public/css${url}`, res);
    } else if (url.indexOf(".png") !== -1) {

```

Create an error handling function

Store the request's URL in a variable url

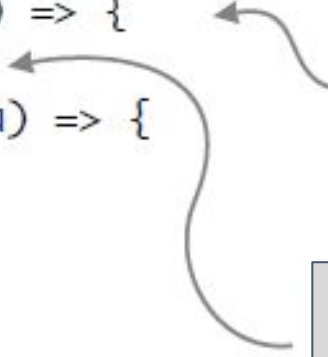
Check the URL to see whether it contains a file extension

Call readFile to read file contents

Customize the response content type

- Upon receiving a request, you need to save its URL in a variable url. With each condition, you need to check url to see whether it contains a file's extension or mime type. You need to customise the response content type to reflect the file being served
- You should put the above repetitive part of the code in a separate function. In the code snippet given, we have named this function **customReadFile**

```
const customReadFile = (file_path, res) => {  
  if (fs.existsSync(file_path)) {  
    fs.readFile(file_path, (error, data) => {  
      if (error) {  
        console.log(error);  
        sendErrorResponse(res);  
        return;  
      }  
      res.write(data);  
      res.end();  
    });  
  } else {  
    sendErrorResponse(res);  
  }  
};
```



Look for a file
by the name
requested

Check whether
the file exists

Poll 3 (15 Sec)

If your application cannot find a route for some request, then you should send back a 404 HTTP status code with a message indicating that the page the client was looking for cannot be found

1. True
2. False

Poll 3 (Answer)

If your application cannot find a route for some request, then you should send back a 404 HTTP status code with a message indicating that the page the client was looking for cannot be found

1. **True**
2. False

Hands-On Exercise 1 (20 mins)

- Create a project folder named `serve_all` and initialise it using ***npm init***
- Install the `http-status-codes` package using ***npm i http-status-codes --S***
- Create the following folders:
 - `views` -> To hold all the HTML pages
 - `public` -> To hold assets
 - subfolders -> `css`, `images`, `js`
- From the previous project, reuse a few HTML files, such as the index page, the aboutus page with pics of team members, the ContactUs page and the thankyou page
- Finally, code the routing logic to handle different file types

Note: The code complexity for routing would reduce once you learn how to use a middleware or a node framework like Express.js. Express.js is a Node web application server framework. Express is the back-end part in the MEAN or MERN stack

[Refer To: serve_all](#)

Core Modules - Events

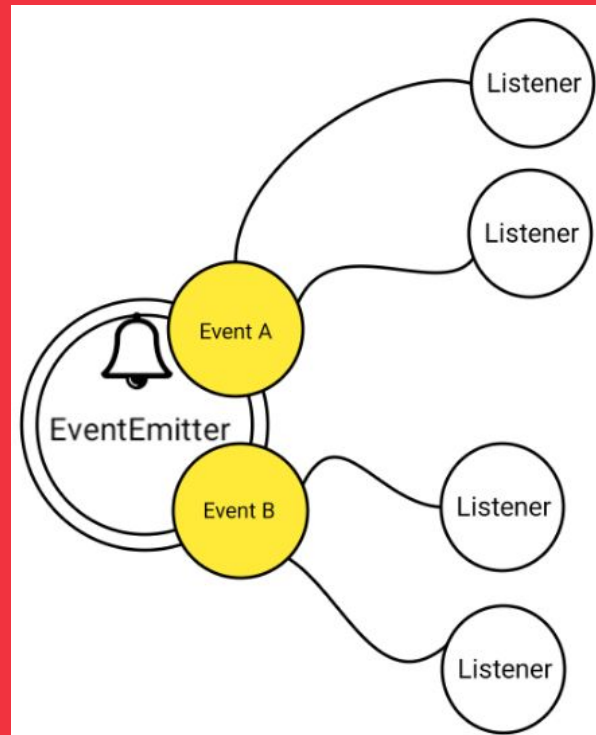
- So far, you have learnt:
 - How to create a web server in Node.js using the HTTP module,
 - How to handle a GET/POST request and response,
 - The application of routing and
 - How to serve a static web application
- Going further, you will learn how different events can be served
- Events on any web application can include:
 - Adding elements to the shopping cart,
 - Checking whether or not a product exists in stock in sufficient quantity at any given time to complete a transaction and
 - Verifying whether a transaction/order has been successfulor any such other events can be served

- An event is an occurrence of something
- Node.js has multiple inbuilt events available through the **events module** and the **EventEmitter class**, which are used to bind events
- You can find the documentation for the **events** module [here](#)
- In JavaScript, the code that listens to an event is written as:

```
<button onclick="this.innerHTML = Date()">Print Date</button>
```

- Similarly, in Node.js, there is an **EventEmitter class**. Event emitters are objects in Node.js that trigger an event by sending a message to signal that an action was completed

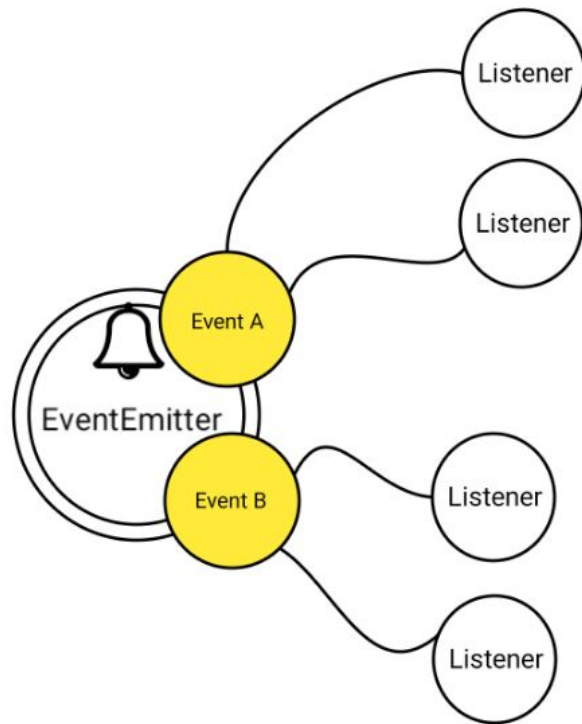
EventEmitter Class



- All objects that emit events are instances of the ***EventEmitter class***
- The EventEmitter class is defined by the events module:

```
const EventEmitter = require('events');
```

- EventEmitter is at the core of the **asynchronous event-driven** architecture of Node.js
- Many of Node's built-in modules inherit from **EventEmitter**, including prominent frameworks like **Express.js**



```
//Import events module
const EventEmitter = require('events');

//Creating an EventEmitter object
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

//bind event with an event handler
myEmitter.on('eventName', eventHandler);

//fire an event
myEmitter.emit('eventName');
```

These are the steps that you need to follow:

1. Import the events module using the keyword require
2. Create an object of the EventEmitter class
3. Bind an event with its eventHandler using the property on
4. Fire an event using the property emit

The working logic of EventEmitter is very simple. Emitter objects emit named events, which cause previously registered listeners to be called. So, an emitter object essentially has two main features:

- Emitting name events
- Registering and unregistering listener functions

Poll 4 (15 Sec)

An event is bound with its eventHandler using the property _____

1. emit
2. require
3. on
4. bind

Poll 4 (Answer)

An event is bound with its eventHandler using the property _____

1. emit
2. require
3. **on**
4. bind

Poll 5 (15 Sec)

How is the EventEmitter class defined?

1. `const EventEmitter = require('eventEmitter');`
2. `const EventEmitter = require('events');`
3. `const EventEmitter = require('EventEmitter');`

Poll 5 (Answer)

How is the EventEmitter class defined?

1. `const EventEmitter = require('eventEmitter');`
2. **`const EventEmitter = require('events');`**
3. `const EventEmitter = require('EventEmitter');`

Example

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

myEmitter.on('event', () => {
  console.log('My event occurred!');
});

myEmitter.emit('event');
```

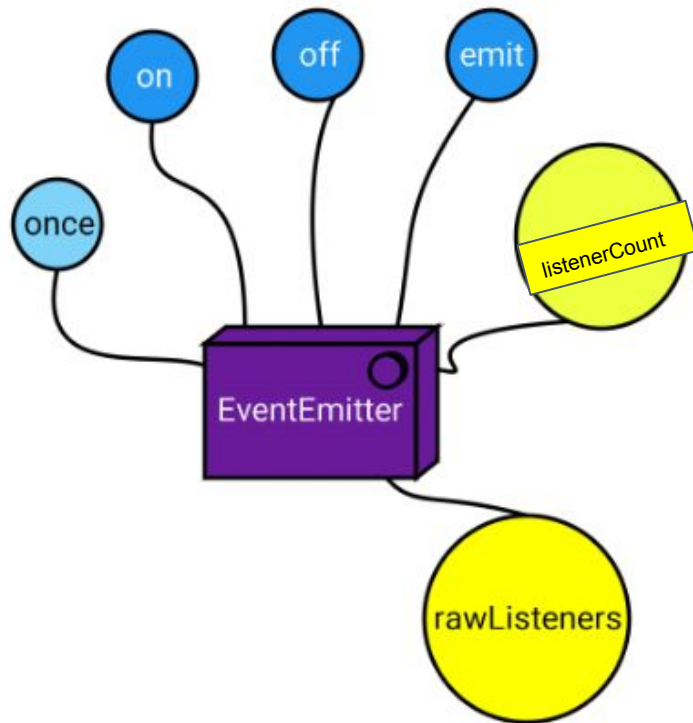
Here is another way of writing the code for the EventEmitter class:

Output

```
Connection successful!  
Data received successfully!  
End of program!
```

```
// Import events module  
const events = require('events');  
  
// Create an object of EventEmitter  
class const EventEmitter = new events.EventEmitter();  
  
// Define connectionMade event's handler  
const connectionHandler = () => {  
  console.log('Connection successful!');  
  // fire dataReceived event  
  EventEmitter.emit('dataReceived');  
};  
  
// bind connectionMade event with its event handler  
EventEmitter.on('connectionMade', connectionHandler);  
  
// bind dataReceived event with its event handler  
EventEmitter.on('dataReceived', () => {  
  console.log('Data received successfully!');  
});  
  
// Fire connectionMade event  
EventEmitter.emit('connectionMade');  
  
// mark end of program  
console.log('End of program!');
```

- The EventEmitter class follows an observer design pattern or a ***publish-subscribe pattern***
- The **APIs** or the **methods** that you will learn about and implement include these:
 - **EventEmitter constructor class** - To create the EventEmitter object
 - **on/addEventListener method** - To add an EventListener to an object
 - **off/removeEventListener method** - To remove the EventListener
 - **once method** - Similar to the method on but listens once only
 - **emit method** - Emits the named event
 - **rawListeners method** - To list the listeners
 - **listenerCount method** - To get the count of listeners



Let's take a look at the following example to understand each of the EventEmitter APIs in detail

- Create an event emitter instance and register more than one callback

Note: on() and addEventListener() are aliases

```
events = require('events');
const myEmitter = new events.EventEmitter();
function handler1() {
  console.log('an event occurred!');
}
function handler2() {
  console.log('yet another event occurred!');
}

myEmitter.on('eventOne', handler1); // Register for eventOne
myEmitter.on('eventOne', handler2); // Register for eventOne

// When the event 'eventOne' is emitted, both the above callbacks should be invoked.
myEmitter.emit('eventOne');
```

Let's take a look at the following example to understand each of the EventEmitter APIs in detail

- Registering for the event to be fired only once using the once method

```
events = require('events');  
  
const myEmitter = new events.EventEmitter();  
  
myEmitter.once('eventOnce', () => console.log('eventOnce once fired'));  
  
myEmitter.emit('eventOnce');  
myEmitter.emit('eventOnce'); // We won't see any o/p for this line of code
```

Let's take a look at the following example to understand each of the EventEmitter APIs in detail

- Registering for the event with callback parameters

```
events = require('events');

const myEmitter = new events.EventEmitter();

myEmitter.on('status', (code, msg) => console.log(`Got ${code} and ${msg}`));

// Emitting the event with parameters
myEmitter.emit('status', 200, 'ok');
```

Let's take a look at the following example to understand each of the EventEmitter APIs in detail

- Unregistering events

Note: off() and removeEventListener() are aliases

```
events = require('events');
const myEmitter = new events.EventEmitter();
function handler1() {
  console.log('an event occurred!');
}
function handler2() {
  console.log('yet another event occurred!');
}

myEmitter.on('eventOne', handler1); // Register for eventOne
myEmitter.on('eventOne', handler2); // Register for eventOne
myEmitter.emit('eventOne'); //We get two outputs
myEmitter.off('eventOne', handler1);
// Now if you emit the event as follows, only handler2 would respond
myEmitter.emit('eventOne');
```


Let's take a look at the following example to understand each of the EventEmitter APIs in detail

- Getting listener count

Note: If the event has been unregistered using the off() or the removeListener() method, then the count will be 0

```
events = require('events');
const myEmitter = new events.EventEmitter();
function handler1() {
  console.log('an event occurred!');
}
function handler2() {
  console.log('yet another event occurred!');
}

myEmitter.on('eventOne', handler1); // Register for eventOne
myEmitter.on('eventOne', handler2); // Register for eventOne
myEmitter.emit('eventOne'); //We get two outputs
myEmitter.off('eventOne', handler1);
// Now if you emit the event as follows, only handler2 would respond
myEmitter.emit('eventOne');
console.log(myEmitter.listenerCount('eventOne'));
```

Let's take a look at the following example to understand each of the EventEmitter APIs in detail

- raw listeners method returns count of active listeners

```
events = require('events');
const myEmitter = new events.EventEmitter();
function handler1() {
  console.log('an event occurred!');
}
function handler2() {
  console.log('yet another event occurred!');
}

myEmitter.on('eventOne', handler1); // Register for eventOne
myEmitter.on('eventOne', handler2); // Register for eventOne
myEmitter.emit('eventOne'); //We get two outputs
myEmitter.off('eventOne', handler1);
// Now if you emit the event as follows, only handler2 would respond
myEmitter.emit('eventOne');
console.log(myEmitter.rawListeners('eventOne'));
```

Poll 6 (15 Sec)

Event emitters follow the publish–subscribe pattern. In this software architecture pattern, a publisher (the event emitter) sends a message (an event) and a subscriber receives the event and performs an action. The power of this pattern is that the publisher does not need to know about the subscribers. A publisher publishes a message, and it is up to the subscribers to react to it in their respective ways. If you wanted to change the behaviour of your application, then you could adjust how the subscribers reacted to the events, without having to change the publisher

True or False?

1. True
2. False

Poll 6 (Answer)

Event emitters follow the publish–subscribe pattern. In this software architecture pattern, a publisher (the event emitter) sends a message (an event) and a subscriber receives the event and performs an action. The power of this pattern is that the publisher does not need to know about the subscribers. A publisher publishes a message, and it is up to the subscribers to react to it in their respective ways. If you wanted to change the behaviour of your application, then you could adjust how the subscribers reacted to the events, without having to change the publisher

True or False?

1. **True**

2. False

Poll 7 (15 Sec)

What will the following code output?

1. Node.js
2. Node.js
MongoDB
3. MongoDB
Express.js
4. Node.js
MongoDB
Express.js

```
events = require('events');  
const myEmitter = new events.EventEmitter();  
  
myEmitter.once('eventOnce', () =>  
  console.log('Node.js')  
);  
  
myEmitter.emit('MongoDB');  
myEmitter.emit('Express.js');  
myEmitter.emit(ReactJS');
```

Poll 7 (Answer)

What will the following code output?

1. Node.js
2. **Node.js**
MongoDB
3. MongoDB
Express.js
4. Node.js
MongoDB
Express.js

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

myEmitter.once('eventOnce', () =>
  console.log('Node.js')
);

myEmitter.emit('MongoDB');
myEmitter.emit('Express.js');
myEmitter.emit('ReactJS');
```

Hands-On Exercise 2 (20 mins)

- Create an event listener for a TicketManager class, which extends the EventEmitter class. This TicketManager class allows a user to buy tickets

***Note 1:** The TicketManager class extends the EventEmitter class. This means the TicketManager class inherits the methods and properties of the EventEmitter class. This is how it gets access to the method emit()*

***Note 2:** In our ticket manager, we would provide the initial supply of tickets that can be purchased*

- Set up listeners for the buy event, which would be triggered every time a ticket is purchased
- Ensure no more than available tickets are being sold
- Code a simple dummy email service and a database service

***Note:** We are calling it dummy because for emailing and database operation, you just log a message on the console*

Doubt Clearance (5 mins)

Homework

Code a **countDown** timer using **EventEmitter** with three listeners:

- One listener will update the user at each second (update)
- The second listener will notify the user when the countdown is nearing its end (end)
- The third listener will trigger 2 seconds before the countdown is complete (end-soon)

Expected output for a countDown of 5 seconds

1 second has passed since the timer started
2 seconds have passed since the timer started
3 seconds have passed since the timer started
Countdown will end in 2 seconds
4 seconds have passed since the timer started
5 seconds have passed since the timer started
Countdown is complete

Refer To: `countDown_eventemitter`

Key Takeaways

- You learnt how to route dynamic web applications

Following tasks are to be completed after today's session

Homework
MCQs
Coding Questions
Course Project - Checkpoint 3

In the next class...

- We will briefly discuss MongoDB and its applications



Thank you!