Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

Chapter 13

Server-Side

Development 2: Node.js

Pearson

In this chapter you will learn . . .

- What is Node.js and how does it differ from PHP
- Node’s unique advantages and disadvantages
- How to use Node to create a REST API with CRUD functionality
- How to use Node with WebSockets to create push-based functionality
- How to use a View Engine to provide a PHP-like developer experience
- What is serverless computing

Introducing Node.js

Node is like PHP in that a Node application can generate HTML in response to HTTP requests, except it uses JavaScript as its programming language.

Node Advantages

- JavaScript Everywhere
- Push Architectures
- Nonblocking Architectures
- Rich Ecosystem of Tools and Code
- Broad Adoption

JavaScript Everywhere

Using the same language on both the client and the server has multiple benefits.

Developer productivity is likely to be higher when there is only a single language to use for the entirety of a project.

With a single language, there are also more opportunities for code sharing and reuse when only a single language is being used.

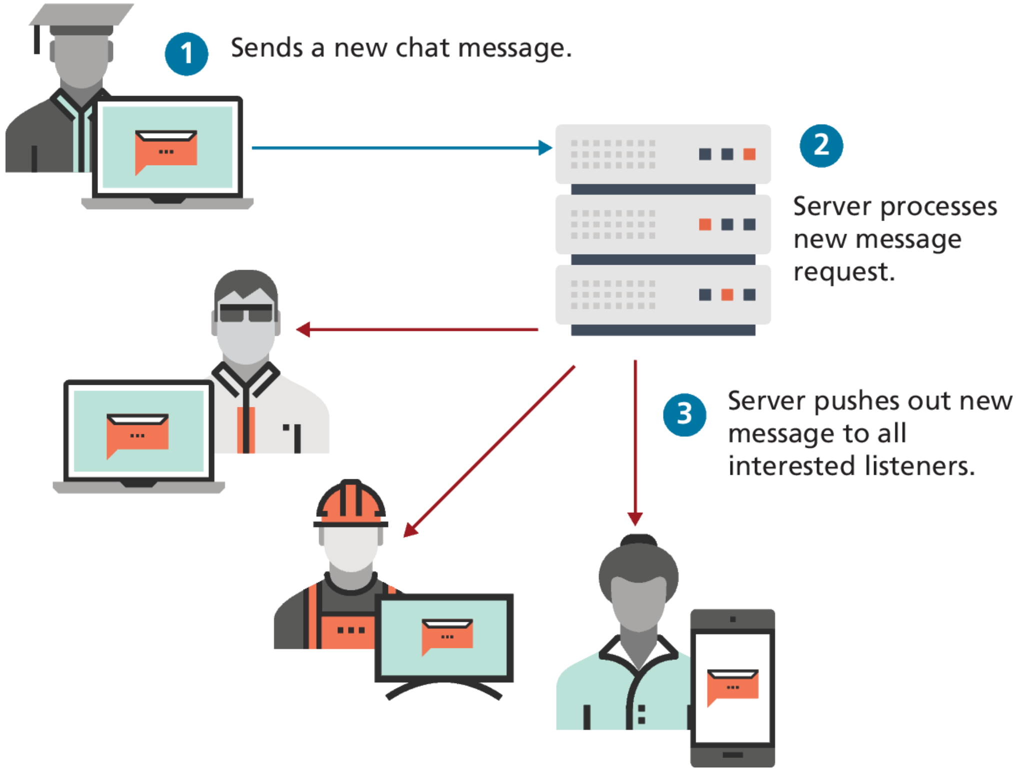
Finally, JavaScript has arguably become the most popular and widely used programming language in the world; this means hiring knowledgeable developers is likely to be easier and that the hiring team only needs to test its potential applicants for knowledge with a single language.

Push Architectures

push-based web applications push information from the server to the client.

Phone calls are push-based: the master phone system pushes out a message (incoming call) to the phone and it responds (by ringing).

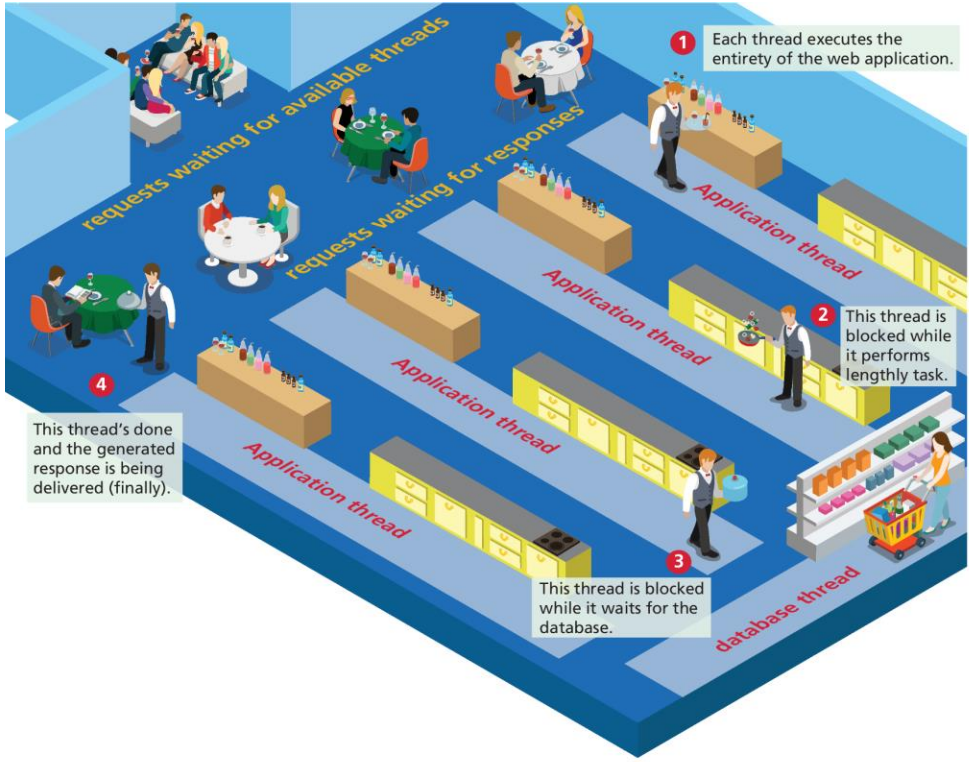
The classic example of a push web application is a chat facility housed within a web page.


1 Sends a new chat message.
2 Server processes new message request.
3 Server pushes out new message to all interested listeners.

Blocking Architectures

Apache runs applications like PHP using either a blocking multiprocessing or multithreaded model.

Using a restaurant analogy, it would be as though a single person had to handle all the tasks required for each table.

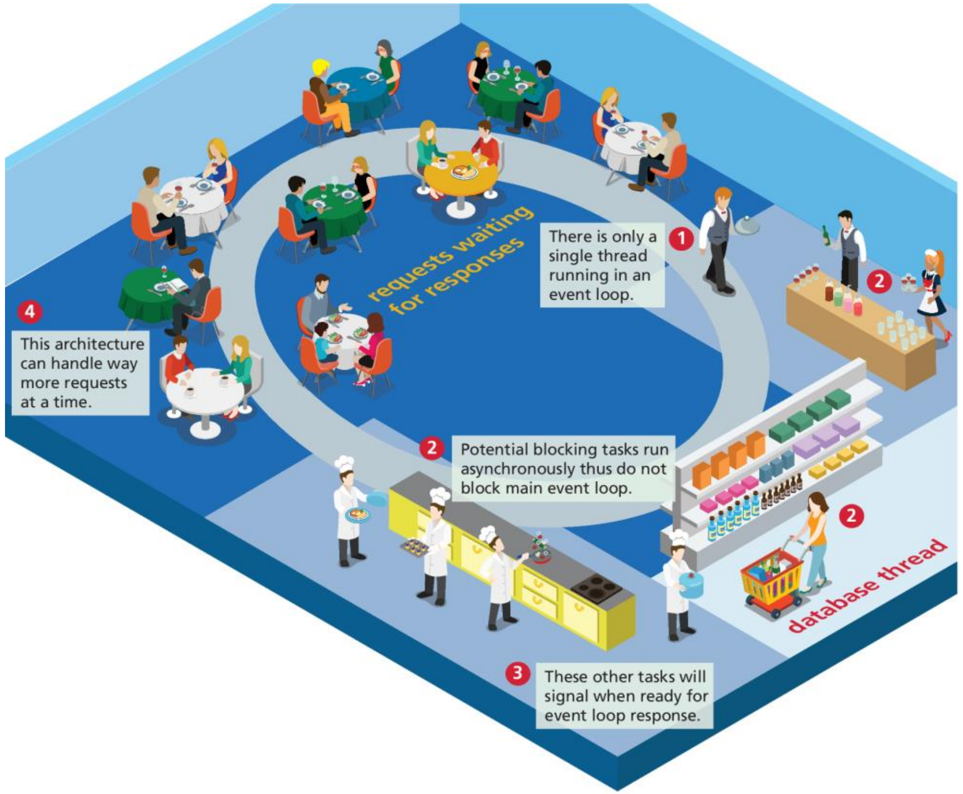

requests waiting for available threads
requests waiting for responses
1 Each thread executes the entirety of the web application.
Application thread
Application thread
Application thread
Application thread
2 This thread is blocked while it performs lengthly task.
4 This thread's done and the generated response is being delivered (finally).
3 This thread is blocked while it waits for the database.
database thread

Nonblocking Architectures

Here a single worker is servicing all the requests in a single event loop thread.

This worker can only be doing a single thing a time. But other tasks are delegated to other agents.

Node uses a **nonblocking**, asynchronous, single-threaded architecture.


requests waiting for responses
1 There is only a single thread running in an event loop.
2
2 Potential blocking tasks run asynchronously thus do not block main event loop.
2
database thread
3 These other tasks will signal when ready for event loop response.
4 This architecture can handle way more requests at a time.

Rich Ecosystem and Broad Adoption

Companies from startups to large companies such as eBay, Netflix, Mozilla, GoDaddy, Groupon, Yahoo, Uber, PayPal, and LinkedIn are using

After more than a decade, Node now has an amazingly rich ecosystem of both prebuilt code and tools that make use of Node.

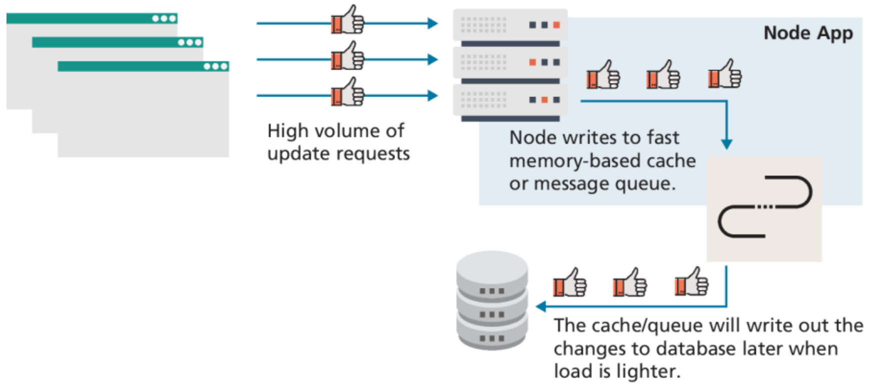
Most new server-based environments either depend on Node or provide Node bindings or APIs. This means that if you want to make use of emerging web development approaches—such as microservices, serverless computing, the Internet of Things, or cloud-service integration—you will find that Node is often a necessity.

Handling high volume data changes in Node

Node is well suited to developing data-intensive real-time applications that need to interact with distributed computers and whose data sources are noSQL databases.

Consider a simple Like button. It would need to handle a massive number of concurrent data writes.

A Node-based system could make use of a memory-based message queueing system that would keep a record of all data changes, and then those changes could eventually be persisted


Node App
High volume of update requests
Node writes to fast memory-based cache or message queue.
The cache/queue will write out the changes to database later when load is lighter.

Node Disadvantages

Node isn’t ideal for all web-based applications. For sites whose data is in a traditional relational database such as MySQL, accessing that data in Node is often a complex programming task.

The single-thread nonblocking architecture of Node is also not ideal for computationally heavy tasks, such as video processing or scientific computing

Applications in Node can be more significantly more complex to develop and maintain than in PHP or in client-side JavaScript

Simple Node Application

```
// make use of the http module
const http = require('http’);

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

// Listen on port 8080 on localhost
const port = 8080;
server.listen(port);

// display a message on the terminal
console.log("Server running at port=" + port);
```

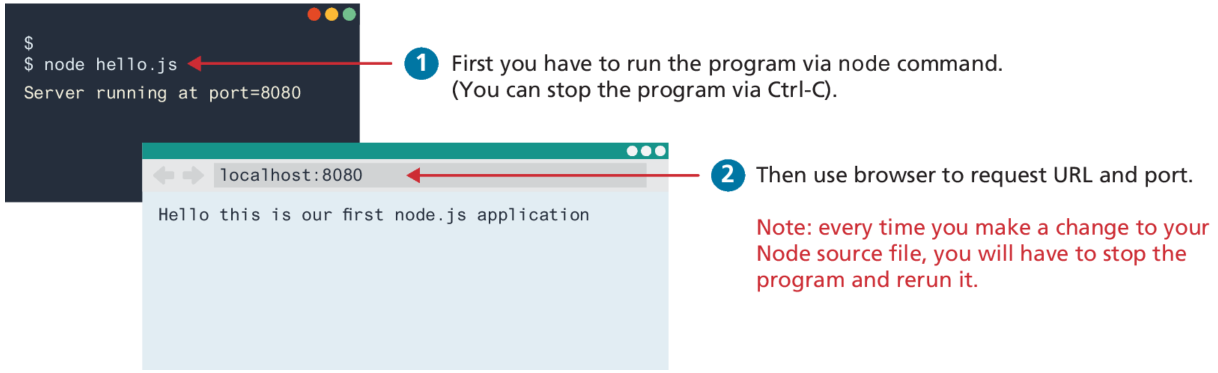
LISTING 13.1 Hello World in Node

Running the Hello World example

If you have installed Node, you will need to open a command/terminal window, navigate to the folder for hello.js , and enter the following command:

```
node hello.js
```

In a browser, make a request for localhost on port 8080.



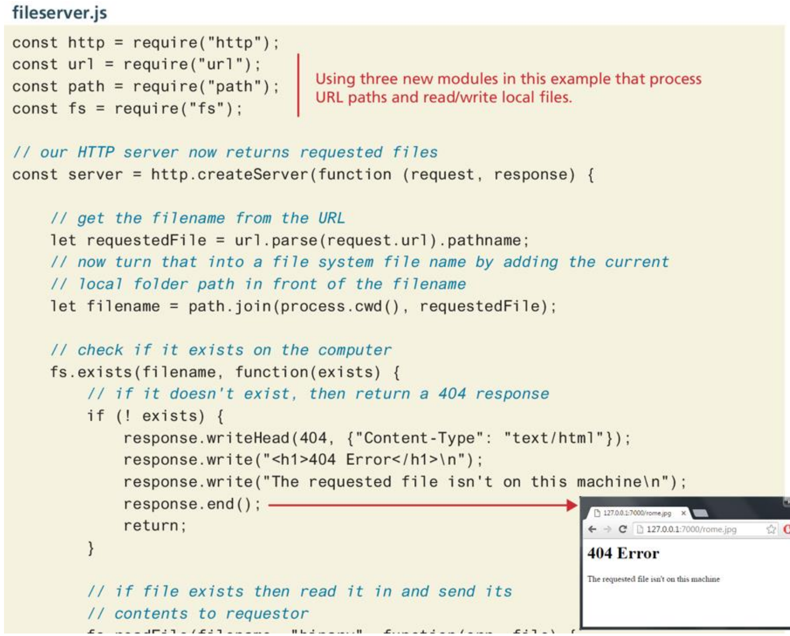
```
$
$ node hello.js
Server running at port=8080
```

```
localhost:8080
```

```
Hello this is our first node.js application
```

1 First you have to run the program via node command. (You can stop the program via Ctrl-C).
2 Then use browser to request URL and port.
Note: every time you make a change to your Node source file, you will have to stop the program and rerun it.

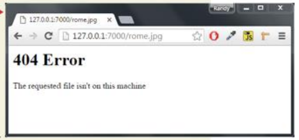
Static file server example



fileserver.js

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

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

    // get the filename from the URL
    let 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
    let 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 file exists then read it in and send its
        // contents to requestor
        fs.readFile(filename, "binary", function(err, file) {
```

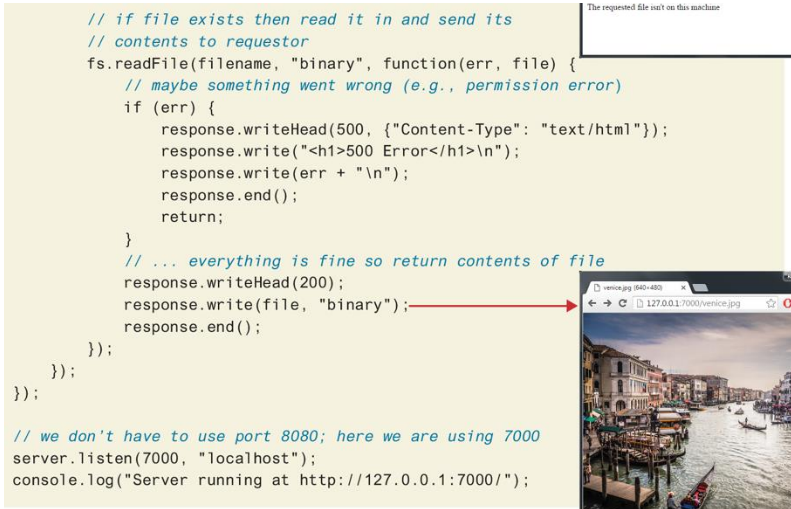
Using three new modules in this example that process URL paths and read/write local files.



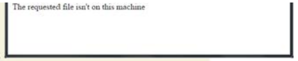
127.0.0.1:7000/rome.jpg

404 Error

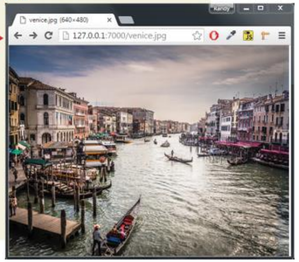
The requested file isn't on this machine



```
        // if file exists then read it in and send its
        // contents to requestor
        fs.readFile(filename, "binary", function(err, file) {
            // maybe something went wrong (e.g., permission error)
            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();
        });
    });
});

// we don't have to use port 8080; here we are using 7000
server.listen(7000, "localhost");
console.log("Server running at http://127.0.0.1:7000/");
```



The requested file isn't on this machine



Adding Express

One of the most popular preexisting modules is **Express**, where you typically write handlers for the different routes in your application. A **route** is a URL, a series of folders or files or parameter data within the URL.

The request object contains a params object that holds any parameter data included with the request/route. The response object provides methods for setting HTTP headers and cookies, sending files or JSON data, and so on. A simple file server using the static() function:

```
var express = require("express");
var app = express();
app.use("/static", express.static(path.join(__dirname, "public")));
app.listen(8080, () => { ... });
```

A middleware function

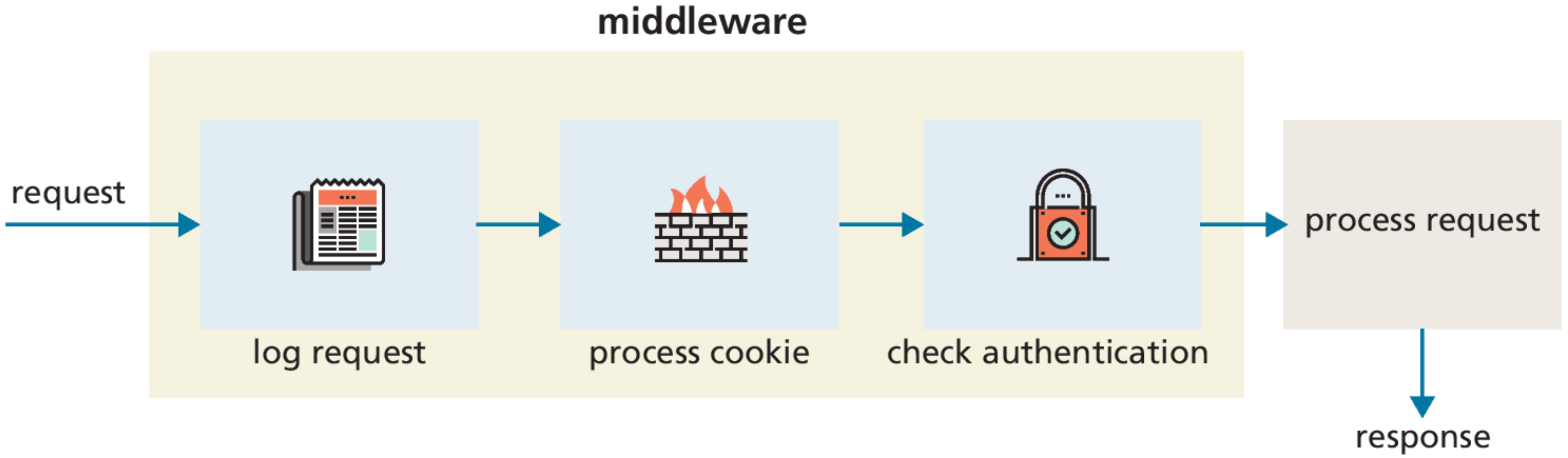
One of the most popular preexisting modules is **Express**, where you typically write handlers for the different routes in your application. A **route** is a URL, a series of folders or files or parameter data within the URL.

The request object contains a params object. The response object provides methods for setting HTTP headers and cookies, sending files or JSON data, and so on. A simple file server using the **static()** function would be:

```
var express = require("express");
var app = express();
app.use("/static", express.static(path.join(__dirname, "public")));
app.listen(8080, () => { ... });
```

The **app.use()** function executes the provided **middleware** function

Middleware functions in Express



Environment Variables

Environment variables provide a mechanism for configuring your Node application. To see your variables:

```
console.log(process.env);
```

You can set your environment variables using the popular dotenv package. Use an environment file named **.env** to store any number of name=value pairs, such as:

```
PORT=8080
BUILD=development
```

Within your Node applications, you can reference the values in this file

```
// make use of dotenv package
require('dotenv').config();
// reference values from the .env file
console.log("build type=" +
process.env.BUILD);
server.listen(process.env.PORT);
```

Creating an API in Node

In Chapters 10 and 11, you consumed external APIs using fetch.

You might have wondered how these APIs were created. While you could use any server-side technology to implement an API, Node is a particularly popular technology for doing so.

Most REST APIs are HTTP front-ends for querying a database. As such, you will learn how to create a database-driven API in the next chapter on databases.

Simple API

Listing 13.3 provides the code for a very simple API that reads in a JSON data file and then returns the JSON data when the URL is requested.

```
// first reference required modules
const fs = require('fs');
const path = require('path');
const express = require('express');
const app = express();
// for now, we will read a json file from public folder
const jsonPath = path.join(__dirname, 'public', 'companies.json’);
// get data using conventional Node callback approach
let companies;
fs.readFile(jsonPath, (err,data) => {
        if (err)
                console.log('Unable to read json data file’);
        else
                companies = JSON.parse(data);
});

// return all the companies when a root request arrives
app.get('/', (req,resp) => { resp.json(companies) } );
// Use express to listen to port
let port = 8080;
app.listen(port, () => {
        console.log("Server running at port= " + port);
});
```

LISTING 13.3 Simple API using callback approach

Adding Routes

To make the web service created in the previous section more useful, let’s add some routes

/companies/:symbol for a single company object

/companies/name/:substring, which will return all companies whose name contains the supplied substring.

```
// return all the companies if a root request arrives
app.get('/', (req,resp) => { resp.json(companies) } );
// return just the requested company, e.g., /companies/amzn
app.get('/companies/:symbol', (req,resp) => {
    const symbolToFind = req.params.symbol.toUpperCase();
    const matches =
    companies.filter(obj => symbolToFind === obj.symbol);
    resp.json(matches);
});
// return companies whose name contains the supplied text,
app.get('/companies/name/:substring', (req,resp) => {
    const substring = req.params.substring.toLowerCase();
    const matches = companies.filter( (obj) =>
    obj.name.toLowerCase().includes(substring) );
    resp.json(matches);
});
```

LISTING 13.4 Adding routes to API

Separating Functionality into Modules

What if we had five or six or more routes? In such a case, our single Node file would start becoming too complex. A better approach would be to separate out the routing functionality into separate modules.

A module in the traditional CommonJS approach in Node is similar to how you created modules in Chapter 10, except rather than using the JavaScript **export** keyword, you instead set the **export** property of the **module** object.

You could also place your route handler logic into a separate module.

Defining a module

```
...
const fs = require('fs').promises;
// for now, we will get our data by reading the provided json file
const jsonPath = path.join(__dirname, '../public', 'companies.json’);

let companies;
getCompanyData(jsonPath);
async function getCompanyData(jsonPath) {
        try {
                const data = await fs.readFile(jsonPath, "utf-8");
                companies = JSON.parse(data);
        }
        catch (err) { console.log('Error reading ' + jsonPath); }
        }
        function getData() {
                return companies;
        }
// specifies which objects will be available outside of module
module.exports = { getData };
```

LISTING 13.5 Defining a module

Creating a CRUD API

It is common to use different HTTP verbs to signal whether we want to create, retrieve, update, or delete (**CRUD**) data.

It is convention to use GET for retrieve requests, POST for create requests, PUT for update requests, and DELETE for delete requests.

For now, our example here will simply modify the in-memory data

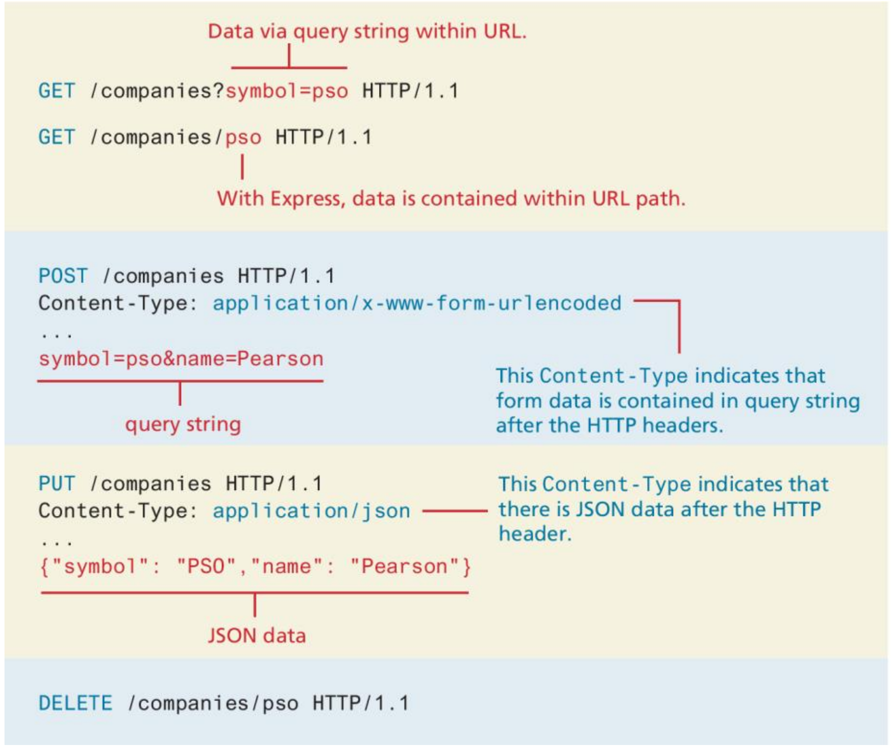
```
const handleSingleSymbol = (companyProvider, app)
=> {
  app.get('/companies/:symbol', (req,resp) => {
      …
  });
  app.put('/companies/:symbol', (req,resp) => {
      …
  });
  app.post('/companies/:symbol', (req,resp) => {
      …
  });
  app.delete('/companies/:symbol', (req,resp) => {
      …
  });
};
```

Passing Data to an API

Recall that form data sent via GET is included via query string parameters added to the URL, while form data sent via POST adds the query string to the request after the HTTP header.

The browser also sets the Content-Type HTTP header to application/x-www-form-urlencoded.

It is possible to also send plain text, JSON or XML data, or file content. To do so requires setting the Content-Type header to the appropriate value.



Data via query string within URL.

```
GET /companies?symbol=pso HTTP/1.1

GET /companies/pso HTTP/1.1
```

With Express, data is contained within URL path.

```
POST /companies HTTP/1.1
Content-Type: application/x-www-form-urlencoded
...
symbol=pso&name=Pearson
```

query string

This Content-Type indicates that form data is contained in query string after the HTTP headers.

```
PUT /companies HTTP/1.1
Content-Type: application/json
...
{"symbol": "PSO","name": "Pearson"}
```

This Content-Type indicates that there is JSON data after the HTTP header.

JSON data

```
DELETE /companies/pso HTTP/1.1
```

API Testing Tools

API developers often make use of some type of third-party API testing tool such as Postman or Insomnia

With these tools you specify the URL endpoint to request, and have full control over which HTTP verbs to use, which Content-Type header to use, and can easily input the data to send.

Working with Web Sockets

WebSockets is an API that makes it possible to open an interactive (two-way) communication channel between the browser and a server that doesn’t use HTTP (except to initiate the communication).

Since Socket.io is not part of the default Node system, to use it you will need to add it to your project via the command:

```
npm install socket.io
```

It consists of two files:

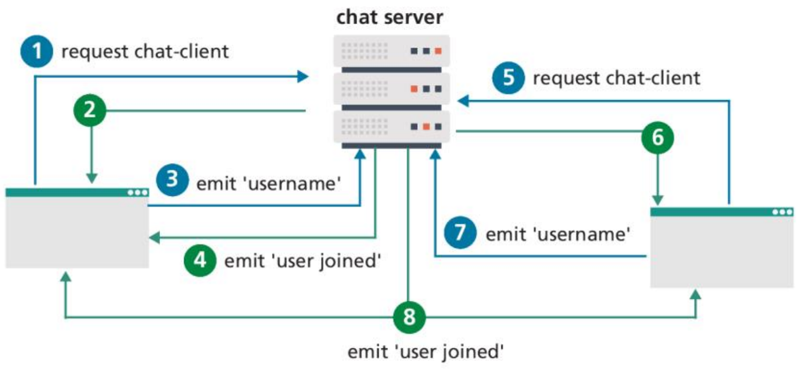
- The Node server application (**chat-server.js**)
- The browser client file (**chat-client.html**)

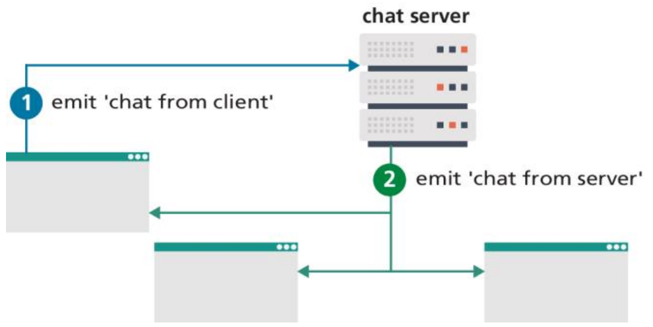
Message flow using Socket.io

The socket.io() function handles the reception of messages from clients.

In Listing 13.9, the server application handles two types of message from its clients: a *username* message and a *chat from client* message. The actual message names can be anything you'd like.

A message is broadcast (or pushed) to all connected clients via the io.emit() function.

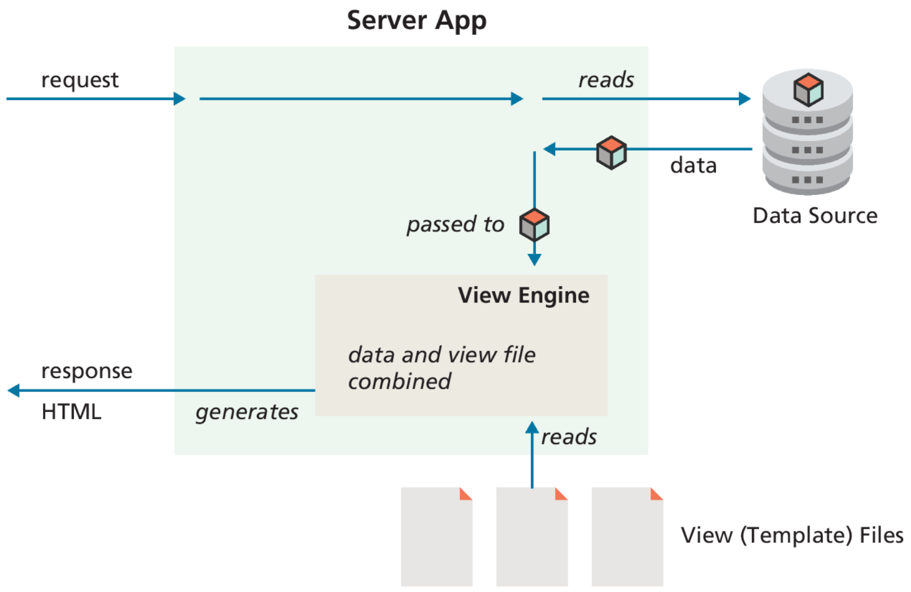




View Engines

It is also possible to use Node in a way similar to PHP: that is, to use JavaScript in Node to generate HTML using a **view engine**

You only need to install the appropriate package using npm, and then tell Express which folder contains the view files and which view engine to use to display those files.



Example EJS view

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>EJS Test</title>
    <link rel="stylesheet" href="/static/list.css" />
</head>
<body><main>
    <h1><%= title %></h1>
    <section class="container">
        <% for (let p of paintings) { %>
            <div class="box">
                <img src="<%= p.filename %>" />
                <h2><%= p.artist %></h2>
                <p><%= p.title %> (<%= p.year %>)</p>
                <button>View</button>
            </div>
        <% } %>
</section> </main> </body> </html>
```

LISTING 13.11 Example EJS view

Serverless Approaches

Serverless computing is a way of deploying and running web applications that make use of third-party services and cloud platforms that replace the need to install, configure, and update back-end environments.

The term serverless is a bit of misnomer in that servers certainly are being used; it is serverless in that you or your team no longer worry.

The serverless approach “is not eliminating complexity, it’s making that complexity someone else’s responsibility.”

Benefits of Serverless Computing

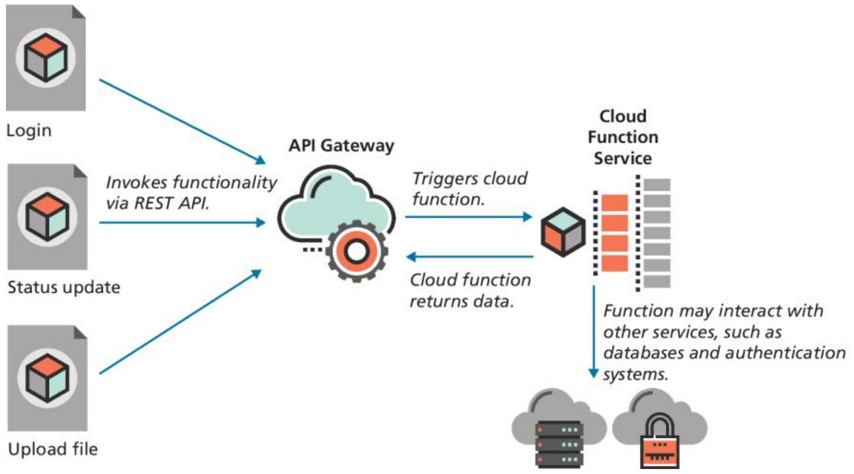
As indicated in the introduction, the main benefit of serverless is that it reduces complexity. There is no need to configure and support server software on your own.

By eliminating the need to provision servers, the serverless approach often results in lower costs.

Serverless doesn’t eliminate servers: it just outsources them to services that can specialize in their support. As such, it generally provides better reliability, scalability, and security.

Serverless Technologies

- Databases-as-a-Service
- Platform-as-a-Service (PaaS)
- Functions-as-a-Service (FaaS)


Login
Status update
Upload file
Invokes functionality via REST API.
API Gateway
Triggers cloud function.
Cloud function returns data.
Cloud Function Service
Function may interact with other services, such as databases and authentication systems.

Key Terms

- CommonJS
- CRUD
- Database-as-a-Service
- Environment variables
- Express
- Functions-as-a-Service
- JAM Stack
- middleware
- module
- Node
- nonblocking
- npm
- Platform-as-a-Service
- push-based
- route
- serverless computing
- templates
- V8
- views
- view engine
- WebSockets

Copyright