

6

Using Node.js Streams

One of the main tasks required in server-side development is transferring data, either reading data sent by a client or browser or writing data that must be transmitted or stored in some way. In this chapter, I will introduce the Node.js API for dealing with data sources and data destinations, known as *streams*. I will explain the concept behind streams, show how they are used to deal with HTTP requests, and explain why one common source of data – the file system – should be used with caution in a server-side project. *Table 6.1* puts streams in context.

Table 6.1: Putting streams in context

Question	Answer
What are they?	Streams are used by Node.js to represent data sources or destinations, including HTTP requests and responses.
Why are they useful?	Streams don't expose the details of how data is produced or consumed, which allows the same code to process data from any source.
How are they used?	Node.js provides streams to deal with HTTP requests. The streams API is used to read data from the HTTP request and write data to the HTTP response.
Are there any pitfalls or limitations?	The streams API can be a little awkward to work with, but this can be improved with the use of third-party packages, which often provide more convenient methods to perform common tasks.
Are there any alternatives?	Streams are integral to Node.js development. Third-party packages can simplify working with streams, but it is helpful to understand how streams work for when problems arise.

Table 6.2 sums up what the chapter will cover.

Table 6.2: Chapter summary

Problem	Solution	Listing
Write data to a stream	Use the <code>write</code> or <code>end</code> methods.	4
Set response headers	Use the <code>setHeader</code> method.	5–7
Manage data buffering	Use the result from the <code>write</code> method and handle the <code>drain</code> event.	8–9
Read data from a stream	Handle the <code>data</code> and <code>end</code> events or use an iterator.	10–15
Connect streams	Use the <code>pipe</code> method.	16
Transform data	Extend the <code>Transform</code> class and use the stream object mode.	17–19
Serve static files	Use the Express static middleware or use the <code>send-File</code> and <code>download</code> methods.	20–26
Encode and decode data	Use the Express JSON middleware and the <code>json</code> response method.	27–28

Preparing for this chapter

In this chapter, I will continue to use the `webapp` project created in *Chapter 4* and modified in *Chapter 3*. To prepare for this chapter, replace the contents of the `server.ts` file in the `src` folder with the code shown in *Listing 6.1*.

Tip



You can download the example project for this chapter – and for all the other chapters in this book – from <https://github.com/PacktPublishing/Mastering-Node.js-Web-Development>. See *Chapter 1* for how to get help if you have problems running the examples.

Listing 6.1: Replacing the contents of the `server.ts` file in the `src` folder

```
import { createServer } from "http";
import express, { Express } from "express";
import { basicHandler } from "../handler";
const port = 5000;
const expressApp: Express = express();
expressApp.get("/favicon.ico", (req, resp) => {
  resp.statusCode = 404;
  resp.end();
});
expressApp.get("*", basicHandler);
const server = createServer(expressApp);
server.listen(port,
  () => console.log(`HTTP Server listening on port ${port}`));
```

The Express router filters out favicon requests and passes on all other HTTP GET requests to a function named `basicHandler`, which is imported from the `handler` module. To define the handler, replace the contents of the `handler.ts` to the `src` folder with the code shown in *Listing 6.2*.

Listing 6.2. The contents of the handler.ts file in the src folder

```
import { IncomingMessage, ServerResponse } from "http";
export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {
  resp.end("Hello, World");
};
```

This handler uses the Node.js `IncomingMessage` and `ServerResponse` types, even though Express is used to route requests. I will demonstrate the enhancements Express provides in the *Using third-party enhancements* section, but I am going to start with the built-in features that Node.js provides.

Some examples in this chapter require an image file. Create the `static` folder and add to it an image file named `city.png`. You can use any PNG image file as long as you name it `city.png`, or you can download the public domain panorama of the New York City skyline that I used, shown in *Figure 6.1*, from the code repository for this chapter.



Figure 6.1: The city.png file in the static folder

Run the command shown in *Listing 6.3* in the `webapp` folder to start the watcher that compiles TypeScript files and executes the JavaScript that is produced.

Listing 6.3: Starting the project

```
npm start
```

Open a web browser and request `http://localhost:5000`. You will see the result shown in *Figure 6.2*.

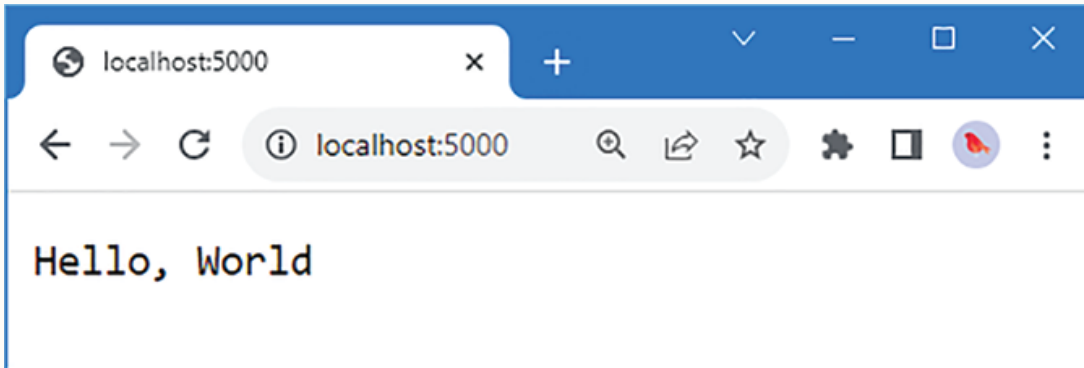


Figure 6.2: Running the example project

Understanding streams

The best way to understand streams is to ignore data and think about water for a moment. Imagine you are in a room in which a pipe with a faucet enters through one wall. Your job is to build a device that will collect the water from the pipe. There is obviously something connected to the other end of the pipe that produces the water, but you are only able to see the faucet, and so the design of your device will be dictated by what you know: you have to create something that will connect to the pipe and receive the water when the faucet is turned on. Having such a limited view of the system you are working with may feel like a restriction, but the pipe can be connected to any source of water and your device works just as well whether the water comes from a river or a reservoir; it is all just water coming through the pipe via the faucet, and it is always consumed consistently.

At the other end of the pipe, the producer of the water has a pipe into which they pump their water. The water producer can't see what you have attached to the other end of the pipe and does not know how you are going to consume the water. And it doesn't matter, because all the producer has to do is push their water through the pipe, regardless of whether their water will be used to drive a water mill, fill a swimming pool, or run a shower. You can change the device attached to your pipe and nothing would change for the producer, who still keeps pumping water into the same pipe in the same way.

In the world of web development, a *stream* solves the problem of distributing data in the same way that the pipe solves the problem of distributing water. Like a pipe, a stream has two ends. At one end is the data producer, also known as the *writer*, who puts a sequence of data values into the stream. At the other end is the data consumer, also known as the *reader*, who receives the sequence of data values from the stream. The writer and reader each have their own API that allows them to work with the stream, as shown in *Figure 6.3*.

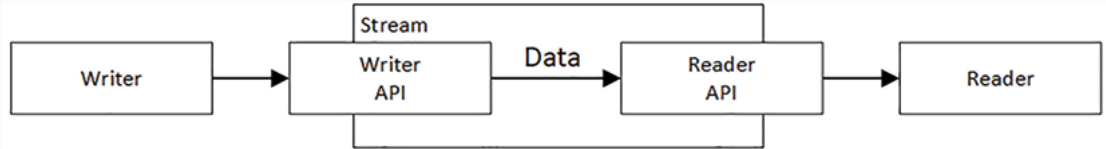


Figure 6.3: The anatomy of a stream

This arrangement has two important characteristics. The first is that the data arrives in the same order in which it is written, which is why streams are usually described as a *sequence* of data values.

The second characteristic is that the data values can be written to the stream over time so that the writer doesn’t have to have all the data values ready before the first value is written. This means that the reader can receive and start processing data while the writer is still preparing or computing later values in the sequence. This makes streams suitable for a wide range of data sources, and they also integrate well with the Node.js programming model, as the examples in this chapter will demonstrate.

Using Node.js streams

The `streams` module contains classes that represent different kinds of streams, and the two most important are described in *Table 6.3*.

Table 6.3: Useful stream classes

Name	Description
<code>Writable</code>	This class provides the API for writing data to a stream.
<code>Readable</code>	This class provides the API for reading data from a stream.

In Node.js development, one end of a stream is usually connected to something outside of the JavaScript environment, such as a network connection or the file system, and this allows data to be read and written in the same way regardless of where it is going to or coming from.

For web development, the most important use of streams is they are used to represent HTTP requests and responses. The `IncomingMessage` and `ServerResponse` classes, which are used to represent HTTP requests and responses, are derived from the `Readable` and `Writable` classes.

Writing data to a stream

The `writable` class is used to write data to a stream. The most useful features provided by the `writable` class are described in *Table 6.4* and explained in the sections that follow.

Table 6.4: Useful Writable Features

Name	Description
<code>write(data, callback)</code>	<p>This method writes data to the stream and invokes the optional callback function when the data has been flushed. Data can be expressed as a <code>string</code>, <code>Buffer</code>, or <code>Uint8Array</code>. For string values, an optional encoding can be specified.</p> <p>The method returns a <code>boolean</code> value that indicates whether the stream is able to accept further data without exceeding its buffer size, as described in the <i>Avoiding excessive data buffering</i> section.</p>
<code>end(data, callback)</code>	<p>This method tells Node.js that no further data will be sent. The arguments are an optional final chunk of data to write and an optional callback function that will be invoked when the data is finished.</p>
<code>destroy(error)</code>	<p>This method destroys the stream immediately, without waiting for any pending data to be processed.</p>
<code>closed</code>	<p>This property returns <code>true</code> if the stream has been closed.</p>
<code>destroyed</code>	<p>This property returns <code>true</code> if the <code>destroy</code> method has been called.</p>
<code>writable</code>	<p>This property returns <code>true</code> if the stream can be written to, meaning that the stream has not ended, encountered an error, or been destroyed.</p>
<code>writableEnded</code>	<p>This property returns <code>true</code> if the <code>end</code> method has been called.</p>

`writable` This property returns the size of the data buffer in bytes. The `write` method will return `false` when the amount of buffered data exceeds this amount.

`errored` This property returns `true` if the stream has encountered an error.

The `writable` class also emits events, the most useful of which are described in *Table 6.5*.

Table 6.5: Useful Writable Events

Name	Description
<code>close</code>	This event is emitted when the stream is closed.
<code>drain</code>	This event is emitted when the stream can accept data without buffering.
<code>error</code>	This event is emitted when an error occurs.
<code>finish</code>	This event is emitted when the <code>end</code> method is called and all of the data in the stream has been processed.

The basic approach to using a writable stream is to call the `write` method until all of the data has been sent to the stream, and then call the `end` method, as shown in *Listing 6.4*.

Listing 6.4: Writing Data in the handler.ts File in the src Folder

```
import { IncomingMessage, ServerResponse } from "http";
export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {
  for (let i = 0; i < 10; i++) {
    resp.write(`Message: ${i}\n`);
  }

  resp.end("End");
};
```

Save the changes, allow Node.js to restart, and then request `http://localhost:5000`. The handler will write its data to the response stream, producing the result shown in *Figure 6.4*.

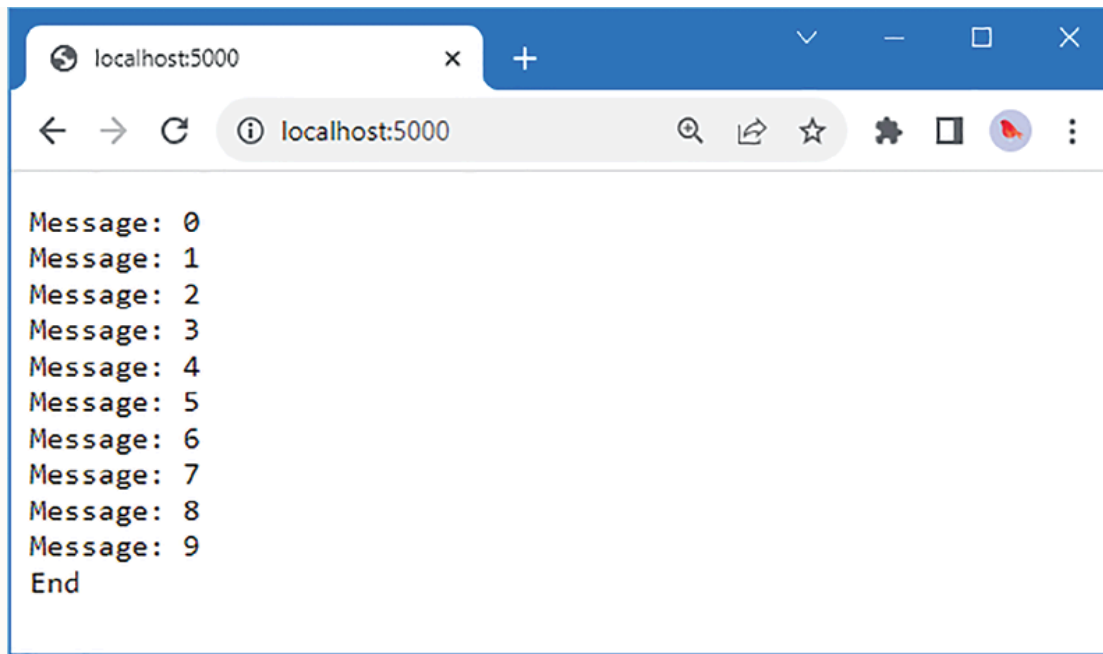


Figure 6.4: Writing data to an HTTP response stream

It is easy to think of the endpoint of the stream as being a straight pipe to the ultimate recipient of the data, which is the web browser in this case, but that's rarely the case. The endpoint for most streams is the part of the Node.js API that interfaces with the operating system, in this case, the code that deals with the operating system's network stack to send and receive data. This indirect relationship leads to important considerations, as described in the sections that follow.

Understanding stream enhancements

Some streams are enhanced to ease development, which means that the data you write to the stream won't always be the data that is received at the other end. In the case of HTTP responses, for example, the Node.js HTTP API aids development by ensuring that all responses conform to the basic requirements of the HTTP protocol, even when the programmer doesn't explicitly use the features provided to set the status code and headers. To see the content that the example in *Listing 6.4* writes to the stream, open a new command prompt and run the Linux command shown in *Listing 6.5*.

Listing 6.5: Making an HTTP Request (Linux)

```
curl --include http://localhost:5000
```

If you are a Windows user, use PowerShell to run the command shown in *Listing 6.6* instead.

Listing 6.6: Making an HTTP Request (Windows)


```
(Invoke-WebRequest http://localhost:5000).RawContent
```

These commands make it easy to see the entire response sent by Node.js. The code in *Listing 6.4* uses just the `write` and `end` methods, but the HTTP response will be like this:

```
...
HTTP/1.1 200 OK
Connection: keep-alive
Keep-Alive: timeout=5
Transfer-Encoding: chunked
Date: Wed, 01 Nov 2023 19:46:02 GMT
X-Powered-By: Express
Message: 0
Message: 1
Message: 2
Message: 3
Message: 4
Message: 5
Message: 6
Message: 7
Message: 8
Message: 9
End
...
```

The Node.js HTTP API makes sure the response is legal HTTP by adding an HTTP version number, a status code and message, and a minimal set of headers. This is a useful feature, and it helps illustrate the fact that you cannot assume that the data you write to a stream will be the data that arrives at the other end.

The `ServerResponse` class demonstrates another kind of stream enhancement, which is methods or properties that write content to the stream for you, as shown in *Listing 6.7*.

Listing 6.7: Using a Stream Enhancement Method in the handler.ts File in the src Folder

```
import { IncomingMessage, ServerResponse } from "http";
export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {
  resp.setHeader("Content-Type", "text/plain");
  for (let i = 0; i < 10; i++) {
    resp.write(`Message: ${i}\n`);
  }

  resp.end("End");
};
```

Behind the scenes, the `ServerResponse` class merges the arguments passed to the `setHeader` method with the default content used for responses. The `ServerResponse` class is derived from `Writable` and implements the methods and properties described in *Table 6.4*, but the enhancements make it easier to write content to the stream that is specific to HTTP requests, like setting a header in the response. If you run the commands shown in *Listing 6.6* or *Listing 6.7* again, you will see the effect of calling the `setHeader` method:

```
...
HTTP/1.1 200 OK
Connection: keep-alive
Keep-Alive: timeout=5
Transfer-Encoding: chunked
Content-Type: text/plain
Date: Wed, 01 Nov 2023 21:19:45 GMT
X-Powered-By: Express
...
```

Avoiding excessive data buffering

Writable streams are created with a buffer in which data is stored before it is processed. The buffer is a way of improving performance, by allowing the producer of data to write data to the stream in bursts faster than the stream endpoint can process them.

Each time the stream processes a chunk of data, it is said to have *flushed* the data. When all of the data in the stream's buffer has been processed, the stream buffer is said to have been *drained*. The amount of data that can be stored in the buffer is known as the *high-water mark*.

A writable stream will always accept data, even if it has to increase the size of its buffer, but this is undesirable because it increases the demand for memory that can be required for an extended period while the stream flushes the data it contains.

The ideal approach is to write data to a stream until its buffer is full and then wait until that data is flushed before further data is written. To help achieve this goal, the `write` method returns a `boolean` value that indicates whether the stream can receive more data without expanding its buffer beyond its target high-water mark.

Listing 6.8 uses the value returned by the `write` method to indicate when the stream buffer has reached capacity.

Listing 6.8: Checking Stream Capacity in the handler.ts File in the src Folder

```
import { IncomingMessage, ServerResponse } from "http";
export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {
  resp.setHeader("Content-Type", "text/plain");
  for (let i = 0; i < 10_000; i++) {
    if (resp.write(`Message: ${i}\n`)) {
      console.log("Stream buffer is at capacity");
    }
  }

  resp.end("End");
};
```

You may need to increase the maximum value used by the `for` loop, but for my development PC, rapidly writing 10,000 messages to the stream will reliably reach the stream limits. Use a browser to request `http://localhost:5000`, and you will see messages like these produced by the Node.js console:

```
...
Stream buffer is at capacity
Stream buffer is at capacity
Stream buffer is at capacity
...
```

Writable streams emit the `drain` event when their buffers have been drained, at which point more data can be written. In *Listing 6.9*, data is written to the HTTP response stream until the `write` method returns `false` and then stops writing until the `drain` event is received. (If you want to know when an individual chunk of data is flushed, then you can pass a callback function to the stream's `write` method.)

Listing 6.9: Avoiding Excessive Data Buffering in the handler.ts File in the src Folder

```
import { IncomingMessage, ServerResponse } from "http";
export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {
  resp.setHeader("Content-Type", "text/plain");
  let i = 0;
  let canWrite = true;
  const writeData = () => {
    console.log("Started writing data");
    do {
      canWrite = resp.write(`Message: ${i++}\n`);
    } while (i < 10_000 && canWrite);
    console.log("Buffer is at capacity");
    if (i < 10_000) {
      resp.once("drain", () => {
        console.log("Buffer has been drained");
      });
    }
  };
  writeData();
};
```

```


        writeData();
    });
} else {
    resp.end("End");
}
}
writeData();
};

```

The `writeData` function enters a `do...while` loop that writes data to the stream until the `write` method returns `false`. The `once` method is used to register a handler that will be invoked once when the `drain` event is emitted, and which invokes the `writeData` function to resume writing. Once all of the data has been written, the `end` method is called to finalize the stream.

Avoiding the Early End Pitfall

A common mistake – and one that I make regularly – is to put the call to the `end` method outside of the callback functions that write the data, like this:



```

...
const writeData = () => {
    console.log("Started writing data");
    do {
        canWrite = resp.write(`Message: ${i++}\n`);
    } while (i < 10_000 && canWrite);
    console.log("Buffer is at capacity");
    if (i < 10_000) {
        resp.once("drain", () => {
            console.log("Buffer has been drained");
            writeData();
        });
    }
}
writeData();
resp.end("End");
...

```

The outcome can differ but is usually an error because the callback will invoke the `write` method after the stream has been closed, or not all the data will be written to the stream because the `drain` event won't be emitted. To avoid this mistake, ensure that the `end` method is invoked within the callback function once the data has been written.

Use a browser to request `http://localhost:5000`, and you will see Node.js console messages that show the writing stops as the buffer reaches capacity, resuming once the buffer is drained:

```
...
Started writing data
Buffer is at capacity
Buffer has been drained
Started writing data
...
```

Reading data from a stream

The most important source of data in a web application comes from HTTP request bodies. The example project needs a little preparation so that the client-side code can make an HTTP request with a body. Add a file named `index.html` to the `static` folder with the content shown in *Listing 6.10*.

Listing 6.10: The Contents of the index.html File in the static Folder

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      document.addEventListener('DOMContentLoaded', function() {
        document.getElementById("btn")
          .addEventListener("click", sendReq);
      });
      sendReq = async () => {
        let payload = "";
        for (let i = 0; i < 10_000; i++) {
          payload += `Payload Message: ${i}\n`;
        }
        const response = await fetch("/read", {
          method: "POST", body: payload
        })
        document.getElementById("msg").textContent
          = response.statusText;
        document.getElementById("body").textContent
          = await response.text();
      }
    </script>
  </head>
  <body>
    <button id="btn">Send Request</button>
    <div id="msg"></div>
```

```
<div id="body"></div>
</body>
</html>
```

This is a simple HTML document that contains some JavaScript code. I'll make improvements later in the chapter, including separating the JavaScript and HTML content into separate files, but this is enough to get started. The JavaScript code in *Listing 6.10* uses the browser's Fetch API to send an HTTP POST request with a body that contains 1,000 lines of text. *Listing 6.11* updates the existing request handler so that it responds with the contents of the HTML file.

Listing 6.11: Updating the Handlers in the handler.ts File in the src Folder

```
import { IncomingMessage, ServerResponse } from "http";
import { readFileSync } from "fs";
export const basicHandler = (req: IncomingMessage, resp: ServerResponse) => {
  resp.write(readFileSync("static/index.html"));
  resp.end();
};
```

I use the `readFileSync` function to perform a blocking read of the `index.html` file, which is simple but is not the best way to read files, as I explain later in this chapter. To create a new handler that will be used to read the data sent by the browser, add a file named `readHandler.ts` to the `src` folder with the contents shown in *Listing 6.12*. For the moment, this handler is a placeholder that ends the response without producing any content.

Listing 6.12: The Contents of the readHandler.ts File in the src Folder

```
import { IncomingMessage, ServerResponse } from "http";
export const readHandler = (req: IncomingMessage, resp: ServerResponse) => {
  // TODO - read request body
  resp.end();
}
```

Listing 6.13 completes the preparation by adding a route that matches POST requests and sends them to the new handler.

Listing 6.13: Adding a Route in the server.ts File in the src Folder

```
import { createServer } from "http";
import express, { Express } from "express";
import { basicHandler } from "../handler";
```

```
import { readHandler } from "./readHandler";
const port = 5000;
const expressApp: Express = express();
expressApp.get("/favicon.ico", (req, resp) => {
  resp.statusCode = 404;
  resp.end();
});
expressApp.get("*", basicHandler);
expressApp.post("/read", readHandler);
const server = createServer(expressApp);
server.listen(port,
  () => console.log(`HTTP Server listening on port ${port}`));
```

Use a browser to request `http://localhost:5000` , and you will see the button defined by the HTML document. Click the button, and the browser will send an HTTP POST request and display the status message from the response it receives, as shown in *Figure 6.5*. The content presented by the browser is completely unstyled, but this is enough for now.

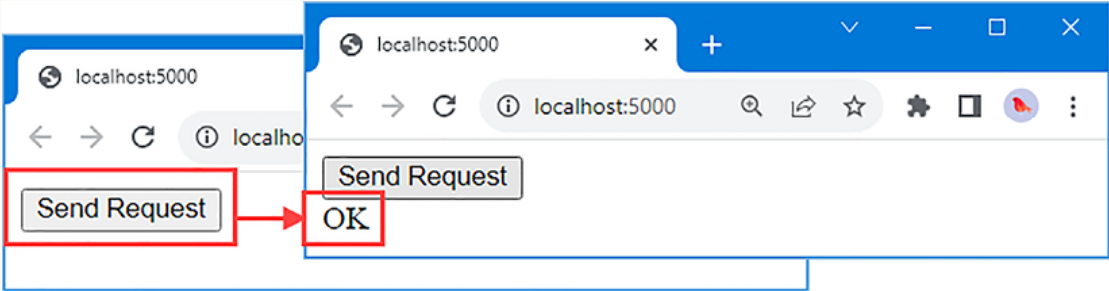


Figure 6.5: Sending an HTTP POST request

Understanding the Readable class

The `Readable` class is used to read data from a stream. *Table 6.6* describes the most useful `Readable` features.

Table 6.6: Useful Readable Features

Name	Description
<code>pause()</code>	Calling this method tells the stream to temporarily stop emitting the <code>data</code> event.
<code>resume()</code>	Calling this method tells the stream to resume emitting the <code>data</code> event.

<code>isPaused()</code>	This method returns <code>true</code> if the stream's <code>data</code> events have been paused.
<code>pipe(writable)</code>	This method is used to transfer the stream's data to a <code>Writable</code> .
<code>destroy(error)</code>	This method destroys the stream immediately, without waiting for any pending data to be processed.
<code>closed</code>	This property returns <code>true</code> if the stream has been closed.
<code>destroyed</code>	This property returns <code>true</code> if the <code>destroy</code> method has been called.
<code>errored</code>	This property returns <code>true</code> if the stream has encountered an error.

The `Readable` class also emits events, the most useful of which are described in *Table 6.7*.

Table 6.7: Useful Readable Events

Name	Description
<code>data</code>	This event is emitted when the stream is in flowing mode and provides access to the data in the stream. See the <i>Reading Data with events</i> section for details.
<code>end</code>	This event is emitted when there is no more data to be read from the stream.
<code>close</code>	This event is emitted when the stream is closed.
<code>pause</code>	This event is emitted when data reading is paused by calling the <code>pause</code> method.
<code>resume</code>	This event is emitted when data reading is restarted by calling the <code>resume</code> method.
<code>error</code>	This event is triggered if there is an error reading data from the stream.

Reading data with events

Data can be read from the stream using events, as shown in *Listing 6.14*, where a callback function is used to process data as it becomes available.

Listing 6.14: Reading Data in the readHandler.ts File in the src Folder

```
import { IncomingMessage, ServerResponse } from "http";
export const readHandler = (req: IncomingMessage, resp: ServerResponse) => {
  req.setEncoding("utf-8");
  req.on("data", (data: string) => {
    console.log(data);
  });
  req.on("end", () => {
    console.log("End: all data read");
    resp.end();
  });
}
```

The `data` event is emitted when data is available to be read from the stream and is available for processing by the callback function used to handle the event. The data is passed to the callback function as a `Buffer`, which represents an array of unsigned bytes, unless the `setEncoding` method has been used to specify character encoding, in which case the data is expressed as a `string`.

This example sets the character encoding to UTF-8 so that the callback function for the `data` event will receive `string` values, which are then written out using the `console.log` method.

The `end` event is emitted when all of the data has been read from the stream. To avoid a variation of the early-end pitfall I described earlier, I call the response's `end` method only when the readable stream's `end` method is emitted. Use a browser to request `http://localhost:5000` and click the **Send Request** button, and you will see a sequence of Node.js console messages as the data is read from the stream:

```
...
Payload Message: 0
Payload Message: 1
Payload Message: 2
Payload Message: 3
...
Payload Message: 9997
Payload Message: 9998
Payload Message: 9999
```

```
End: all data read
...
```

The JavaScript main thread ensures that `data` events are processed sequentially, but the basic idea is that data is read and processed as quickly as possible, such that the `data` event will be emitted as soon as possible once data is available to be read.

Reading data with an iterator

Instances of the `Readable` class can be used as a source of data in a `for` loop, which can provide a more familiar way to read data from a stream, as shown in *Listing 6.15*.

Listing 6.15: Reading Data in a Loop in the `readHandler.ts` File in the `src` Folder

```
import { IncomingMessage, ServerResponse } from "http";
export const readHandler = async (req: IncomingMessage, resp: ServerResponse) => {
  req.setEncoding("utf-8");
  for await (const data of req) {
    console.log(data);
  }
  console.log("End: all data read");
  resp.end();
}
```

The `async` and `await` keywords must be used as shown in the example, but the result is that the `for` loop reads data from the stream until it is all consumed. This example produces the same output as *Listing 6.14*.

Piping data to a writable stream

The `pipe` method is used to connect a `Readable` stream to a `Writable` stream, ensuring that all of the data is read from the `Readable` and written to the `Writable` without further intervention, as shown in *Listing 6.16*.

Listing 6.16: Piping Data into the `readHandler.ts` File in the `src` Folder

```
import { IncomingMessage, ServerResponse } from "http";
export const readHandler = async (req: IncomingMessage, resp: ServerResponse) => {
  req.pipe(resp);
}
```

This is the simplest way to transfer data between streams, and the `end` method is called automatically on the `Writable` stream once all of the data has been transferred. Use a browser to request `http://localhost:5000` and click the **Send Request** button. The

data that is sent in the HTTP request is piped to the HTTP response and displayed in the browser window, as shown in *Figure 6.6*.

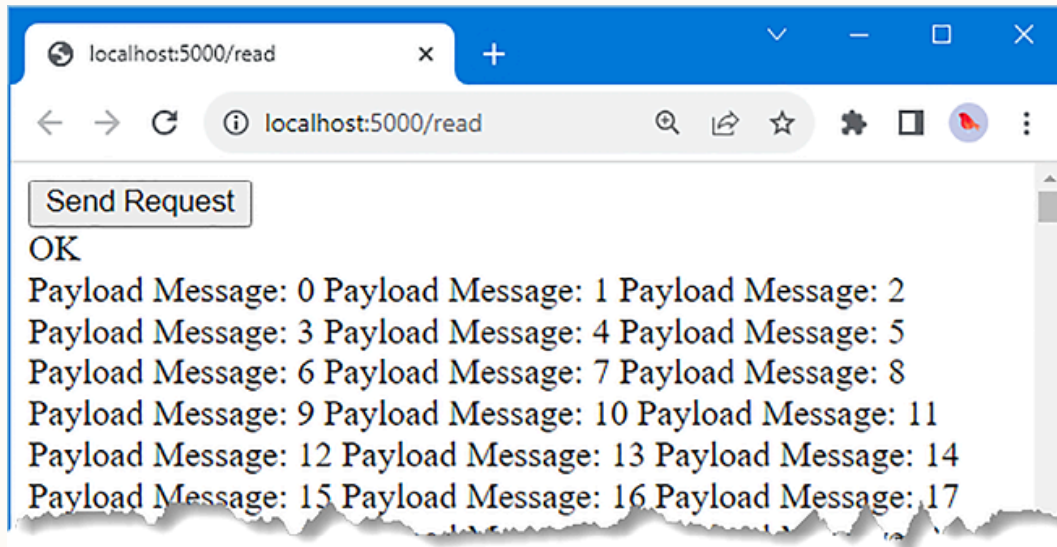


Figure 6.6: Piping data

Transforming data

The `Transform` class is used to create objects, known as *transformers*, that receive data from a `Readable` stream, process it in some way, and then pass it on. Transformers are applied to streams with the `pipe` method, as shown in *Listing 6.17*.

Listing 6.17: Creating a Transformer in the `readHandler.ts` File in the `src` Folder

```
import { IncomingMessage, ServerResponse } from "http";
import { Transform } from "stream";
export const readHandler = async (req: IncomingMessage, resp: ServerResponse) => {
  req.pipe(createLowerTransform()).pipe(resp);
}
const createLowerTransform = () => new Transform({
  transform(data, encoding, callback) {
    callback(null, data.toString().toLowerCase());
  }
});
```

The argument to the `Transform` constructor is an object whose `transform` property value is a function that will be invoked when there is data to process. The function receives three arguments: a chunk of data to process, which can be of any data type, a string encoding type, and a callback function that is used to pass on the transformed data. In this example, the data that is received is converted to a string on which the `toLowerCase`

`case` method is called. The result is passed to the callback function, whose arguments are an object that represents any error that has occurred and the transformed data.

The transformer is applied with the `pipe` method and, in this case, is chained so that the data read from the HTTP request is transformed and then written to the HTTP response. Note that a new `Transform` object must be created for every request, like this:

```
...
req.pipe(createLowerTransform()).pipe(resp);
...
```

Use a browser to request `http://localhost:5000`, and click on the **Send Request** button. The content displayed by the browser, which comes from the HTTP response body, is all lowercase, as shown in *Figure 6.7*.

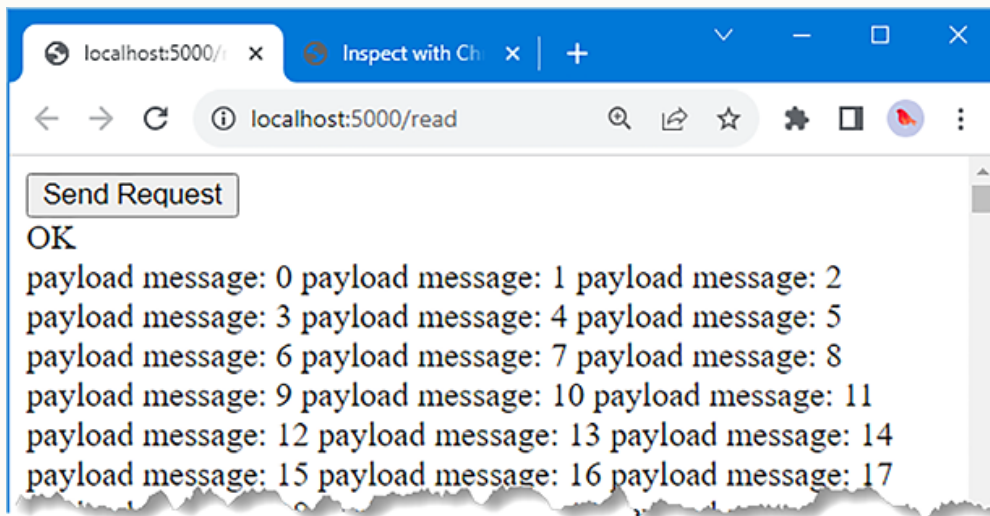


Figure 6.7: Using a simple transformer

Using object mode

The streams created by the Node.js API, such as the ones used for HTTP requests or files, work only on strings and byte arrays. This isn't always convenient, and so some streams, including transformers, can use *object mode*, which allows objects to be read or written. To prepare for this example, *Listing 6.18* updates the JavaScript code contained within the static HTML file to send a request containing an array of JSON-formatted objects.

Listing 6.18: Sending a JSON Request Body in the index.html File in the static Folder

```
...
<script>
  document.addEventListener('DOMContentLoaded', function() {
```

```

        document.getElementById("btn").addEventListener("click", sendReq);
    });
    sendReq = async () => {
        let payload = [];
        for (let i = 0; i < 5; i++) {
            payload.push({ id: i, message: `Payload Message: ${i}\n`});
        }
        const response = await fetch("/read", {
            method: "POST", body: JSON.stringify(payload),
            headers: {
                "Content-Type": "application/json"
            }
        });
        document.getElementById("msg").textContent = response.statusText;
        document.getElementById("body").textContent = await response.text();
    }
</script>
...

```

The data sent by the client can still be read as a string or a byte array, but a transform can be used to convert the request payload into a JavaScript object or convert a JavaScript object into a string or byte array, known as *object mode*. Two `Transform` constructor configuration settings are used to tell Node.js how a transformer will behave, as described in *Table 6.8*.

Table 6.8: The Transform Constructor Configuration Settings

Name	Description
<code>readableObjectMode</code>	When set to <code>true</code> , the transformer will consume string/byte data and produce an object.
<code>writableObjectMode</code>	When set to <code>true</code> , the transformer will consume an object and produce string/byte data.

Listing 6.19 shows a transformer that sets the `readableObjectMode` setting to `true`, which means that it will read string data from the HTTP request payload but produce a JavaScript object when its data is read.

Listing 6.19: Parsing JSON in the readHandler.ts File in the src Folder

```

import { IncomingMessage, ServerResponse } from "http";
import { Transform } from "stream";
export const readHandler = async (req: IncomingMessage, resp: ServerResponse) => {
    if (req.headers["content-type"] == "application/json") {

```

```

    req.pipe(createFromJsonTransform()).on("data", (payload) => {
      if (payload instanceof Array) {
        resp.write(`Received an array with ${payload.length} items`)
      } else {
        resp.write("Did not receive an array");
      }
      resp.end();
    });
  } else {
    req.pipe(resp);
  }
}

const createFromJsonTransform = () => new Transform({
  readableObjectMode: true,
  transform(data, encoding, callback) {
    callback(null, JSON.parse(data));
  }
});

```

If the HTTP request has a `Content-Type` header that indicates the payload is JSON, then the transformer is used to parse the data, which is received by the request handler using the `data` event. The parsed payload is checked to see if it is an array, and if it is, then its length is used to generate a response. Use a browser to request `http://localhost:5000` (or make sure to reload the browser so that the changes in *Listing 6.18* take effect), click the **Send Request** button, and you will see the response shown in *Figure 6.8*.

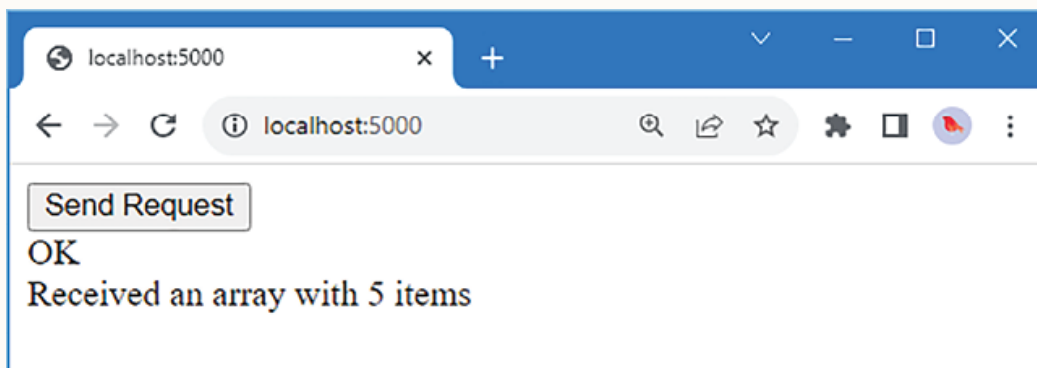


Figure 6.8: Using a transformer in object mode

Using third-party enhancements

In the sections that follow, I describe useful enhancements provided by the Express package to deal with streams and tasks that are related to HTTP. Express isn't the only package that provides these kinds of features, but it is a good default choice for new projects and gives you a foundation from which to compare alternatives.

Working with files

One of the most important tasks for a web server is to respond to requests for files, which provide browsers with the HTML, JavaScript, and other static content required by the client-side part of the application.

Node.js provides a comprehensive API to deal with files in the `fs` module, and it has support for reading and writing streams, along with convenience features that read or write complete files, such as the `readFileSync` function I used to read the contents of an HTML file.

The reason I have not described the API in any detail is that working directly with files within a web server project is incredibly dangerous and should be avoided whenever possible. There is a huge scope to create malicious requests whose paths attempt to access files outside of the expected locations, for example. And, through personal experience, I have learned not to let clients create or modify files on the server under any circumstances.

I have worked on too many projects where malicious requests have been able to overwrite system files or simply overwhelm servers by writing so much data that storage space is exhausted.

The best way to deal with files is to use a well-tested package, rather than write custom code, and it is for this reason that I have not described the features of the `fs` module.



Note

If you decide to ignore my warning, then you can find details of the `fs` module and the features it provides at <https://nodejs.org/dist/latest-v20.x/docs/api/fs.html>.

The Express package has integrated support to serve requests for files. To prepare, add a file named `client.js` to the `static` folder with the content shown in *Listing 6.20*.

Listing 6.20: The Contents of the client.js File in the static Folder

```
document.addEventListener('DOMContentLoaded', function() {
  document.getElementById("btn").addEventListener("click", sendReq);
});
sendReq = async () => {
  let payload = [];
  for (let i = 0; i < 5; i++) {
    payload.push({ id: i, message: `Payload Message: ${i}\n`});
  }
}
```



```
const response = await fetch("/read", {
  method: "POST", body: JSON.stringify(payload),
  headers: {
    "Content-Type": "application/json"
  }
})
document.getElementById("msg").textContent = response.statusText;
document.getElementById("body").textContent = await response.text();
}
```

This is the same JavaScript code used in earlier examples but put into a separate file, which is the typical way of distributing JavaScript to clients. *Listing 6.21* updates the HTML file to link to the new JavaScript file, and it also includes the image file that was added to the project at the start of the chapter.

Listing 6.21: Changing Content in the index.html File in the static Folder

```
<!DOCTYPE html>
<html>
  <head>
    <script src="client.js"></script>
  </head>
  <body>
    
    <button id="btn">Send Request</button>
    <div id="msg"></div>
    <div id="body"></div>
  </body>
</html>
```

Having prepared the content, the next step is to configure Express to serve the files. Express comes with support for middleware components, which just means request handlers that can inspect and intercept all the HTTP requests the server receives. Middleware components are set up with the `use` method, and *Listing 6.22* sets up the middleware component that Express provides to serve files.

Listing 6.22: Adding Support for Static Files in the server.ts File in the src Folder

```
import { createServer } from "http";
import express, { Express } from "express";
//import { basicHandler } from "../handler";
```



```
const expressApp: Express = express();
// expressApp.get("/favicon.ico", (req, resp) => {
```



```
//     resp.statusCode = 404;
//     resp.end();
// });
//expressApp.get("*", basicHandler);
expressApp.post("/read", readHandler);
expressApp.use(express.static("static"));
const server = createServer(expressApp);
server.listen(port,
    () => console.log(`HTTP Server listening on port ${port}`));
```

The `express` object, which is the default export from the `express` module, defines a method named `static` that creates the middleware component that serves static files. The argument to the `static` method is the directory that contains the files, which is also named `static`. The result is a request handler that can be registered with the `Express.use` method.

The middleware component will attempt to match request URLs to files in the `static` directory. The name of the directory that contains the files is omitted from the URLs, so a request for `http://localhost:5000/client.js`, for example, will be handled by returning the contents of the `client.js` file in the `static` folder.

The `static` method can accept a configuration object, but the default values are well-chosen and suit most projects, including using the `index.html` as the default for requests.



Tip

If you need to change the settings, you can see the options at <https://expressjs.com/en/4x/api.html#express.static>.

The middleware component sets the response headers to help the client process the contents of the files that are used. This includes setting the `Content-Length` header to specify the amount of data the file contains, and the `Content-Type` header to specify the type of data.

Notice that I can remove some of the existing handlers from the example. The handler for `favicon.ico` requests is no longer required because the new middleware will automatically generate “not found” responses when requests ask for files that don’t exist. The catch-all route is no longer required because the `static` middleware responds to requests with the contents of the `index.html` file. Use a browser to request `http://localhost:5000`, and you will see the response shown in *Figure 6.9*, which also shows the data types that the browser has received.

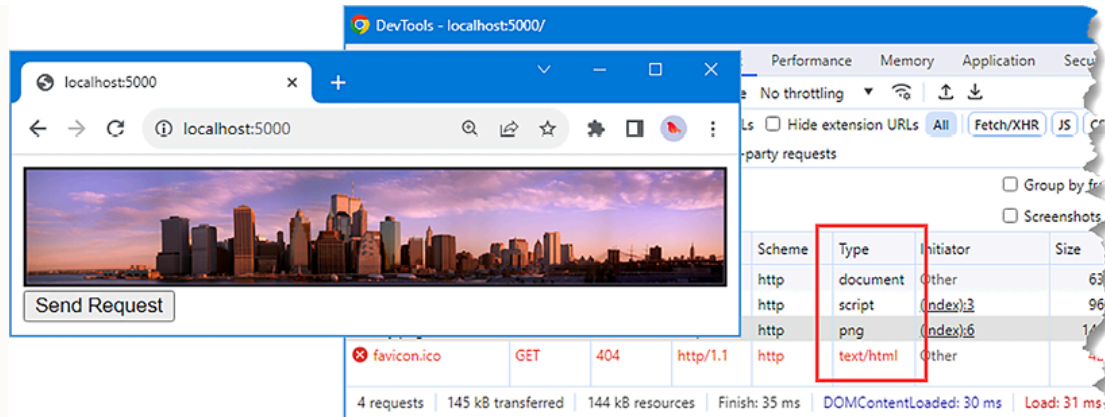


Figure 6.9: Using the Express static middleware

Serving files from client-side packages

One source of static files is packages that are added to the Node.js project, but whose files are intended for consumption by browsers (or other HTTP clients). A good example is the Bootstrap CSS package, which contains CSS stylesheets and JavaScript files that are used to style the HTML content displayed by browsers.

If you are using a client-side framework such as Angular or React, these CSS and JavaScript files will be incorporated into a single compressed file as part of the project build process.

For projects that don't use these frameworks, the simplest way to make the files available is to set up additional instances of the static file middleware. To prepare, run the command shown in *Listing 6.23* in the `webapp` folder to add the Bootstrap package to the example project.

Listing 6.23: Adding a Package to the Example Project

```
npm install bootstrap@5.3.2
```

Listing 6.24 configures Express to serve files from the package directory.

Listing 6.24. Adding Middleware in the `server.ts` File in the `src` Folder

```
import { createServer } from "http";
import express, { Express } from "express";
import { readHandler } from "../readHandler";
const port = 5000;
const expressApp: Express = express();
expressApp.post("/read", readHandler);
expressApp.use(express.static("static"));
```

```
expressApp.use(express.static("node_modules/bootstrap/dist"));
const server = createServer(expressApp);
server.listen(port,
  () => console.log(`HTTP Server listening on port ${port}`));
```

Some knowledge of the packages you are using is required. In the case of the Bootstrap package, I know that the files used by clients are in the `dist` folder, and so this is the folder that I specified when setting up the middleware clients. The final step is to add a reference to a Bootstrap stylesheet and apply the styles it contains, as shown in *Listing 6.25*.

Listing 6.25. Adding a Stylesheet Reference in the index.html File in the static Folder

```
<!DOCTYPE html>
<html>
  <head>
    <script src="client.js"></script>
    <link href="css/bootstrap.min.css" rel="stylesheet" />
  </head>
  <body>
    
    <button id="btn" class="btn btn-primary my-2">Send Request</button>
    <div id="msg"></div>
    <div id="body"></div>
  </body>
</html>
```

The `bootstrap.min.css` file contains the styles I want to use, which are applied by adding the `button` element to classes. Use a browser to request `http://localhost:5000`, and you will see the effect of the styles, as shown in *Figure 6.10*.

Note



See <https://getbootstrap.com> for details of the features the Bootstrap package provides, some of which I use in later chapters. There are other CSS packages available if you can't get along with Bootstrap. A popular alternative is Tailwind (<https://tailwindcss.com>), but a quick web search will present you with a long list of alternatives to consider.

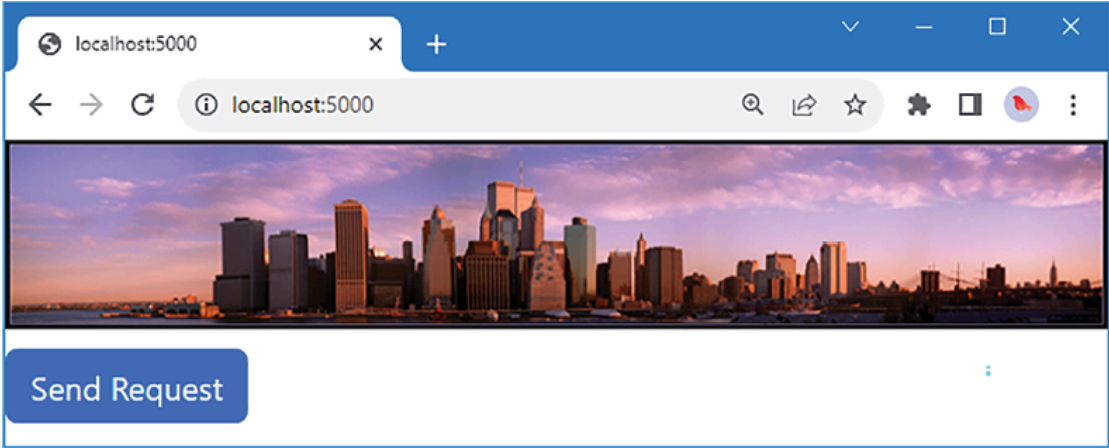


Figure 6.10: Using static content from a third-party package

Sending and downloading files

The `Response` class, through which Express provides `ServerResponse` enhancements, defines the methods described in *Table 6.9* to deal with files directly.

Table 6.9: Useful Response Methods for Files

Name	Description
<code>sendFile(path, config)</code>	This method sends the contents of the specified file. The response <code>Content-Type</code> header is set based on the file extension.
<code>download(path)</code>	This method sends the contents of the specified file such that most browsers will prompt the user to save the file.

The `sendFile` and `download` methods are useful because they provide solutions to problems that cannot be solved using the `static` middleware. *Listing 6.26* creates simple routes that use these methods.

Listing 6.26: Adding Routes in the `server.ts` File in the `src` Folder

```
import { createServer } from "http";
import express, { Express, Request, Response } from "express";
import { readHandler } from "../readHandler";
const port = 5000;
const expressApp: Express = express();
expressApp.post("/read", readHandler);
expressApp.get("/sendcity", (req, resp) => {
  resp.sendFile("city.png", { root: "static" });
});
```

```

expressApp.get("/downloadcity", (req: Request, resp: Response) => {
    resp.download("static/city.png");
});

expressApp.get("/json", (req: Request, resp: Response) => {
    resp.json("{name: Bob}");
});

expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
const server = createServer(expressApp);
server.listen(port,
    () => console.log(`HTTP Server listening on port ${port}`));

```

The `sendFile` method is useful when you need to respond with the content of a file but the request path doesn't contain the filename. The arguments are the name of the file and a configuration object, whose `root` property specifies the directory that contains the file.

The `download` method sets the `Content-Disposition` response header, which causes most browsers to treat the file contents as a download that should be saved. Use a browser to request `http://localhost:5000/sendcity` and `http://localhost:5000/downloadcity`. The first URL will cause the browser to display the image in the browser window. The second URL will prompt the user to save the file. Both responses are shown in *Figure 6.11*.

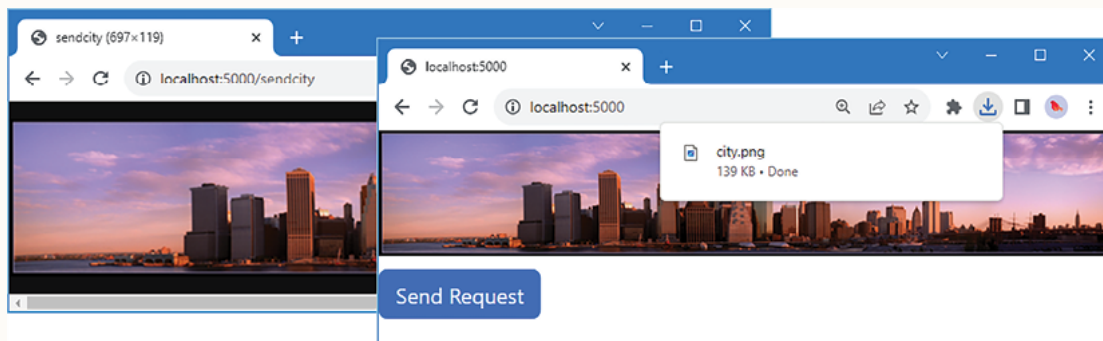


Figure 6.11: Using the file response enhancements

Automatically decoding and encoding JSON

The Express package includes a middleware component that decodes JSON response bodies automatically, performing the same task as the stream transformer I created earlier in the chapter. *Listing 6.27* enables this middleware by calling the `json` method defined on the default export from the `express` module.

Listing 6.27: Enabling JSON Middleware in the server.ts File in the src Folder

```
import { createServer } from "http";
import express, { Express, Request, Response } from "express";
import { readHandler } from "../readHandler";
const port = 5000;
const expressApp: Express = express();
expressApp.use(express.json());
expressApp.post("/read", readHandler);
expressApp.get("/sendcity", (req, resp) => {
    resp.sendFile("city.png", { root: "static" });
});
expressApp.get("/downloadcity", (req: Request, resp: Response) => {
    resp.download("static/city.png");
});
expressApp.get("/json", (req: Request, resp: Response) => {
    resp.json("{name: Bob}");
});

expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
const server = createServer(expressApp);
server.listen(port,
    () => console.log(`HTTP Server listening on port ${port}`));
```

The middleware component must be registered before the routes that read response bodies so that JSON requests are parsed before they are matched to a handler.

Note



The `json` method can accept a configuration object that changes the way that JSON is parsed. The defaults are suitable for most projects, but see <https://expressjs.com/en/4x/api.html#express.json> for details of the available options.

The `Request` class through which Express provides enhancements to the `IncomingRequest` class defines a `body` property, which is assigned the object created by the JSON middleware.

The `Response` body, which provides `ServerResponse` enhancements, defines the `json` method, which accepts an object that is serialized to JSON and used as the response body.

Listing 6.28 updates the handler to use the `Request` class, disables the custom transformer, and sends a JSON response to the client.

Listing 6.28: Using the JSON Object in the readHandler.ts File in the src Folder

```
import { IncomingMessage, ServerResponse } from "http";
//import { Transform } from "stream";
import { Request, Response } from "express";
export const readHandler = async (req: Request, resp: Response) => {
  if (req.headers["content-type"] == "application/json") {
    const payload = req.body;
    if (payload instanceof Array) {
      //resp.write(`Received an array with ${payload.length} items`)
      resp.json({arraySize: payload.length});
    } else {
      resp.write("Did not receive an array");
    }
    resp.end();
  } else {
    req.pipe(resp);
  }
}
// const createFromJsonTransform = () => new Transform({
//   readableObjectMode: true,
//   transform(data, encoding, callback) {
//     callback(null, JSON.parse(data));
//   }
// });
```

Use a web browser to request `http://localhost:5000`, and click the **Send Request** button. The response will confirm that the JSON request body was parsed into a JavaScript array and the response was sent back as JSON as well, as shown in *Figure 6.12*.

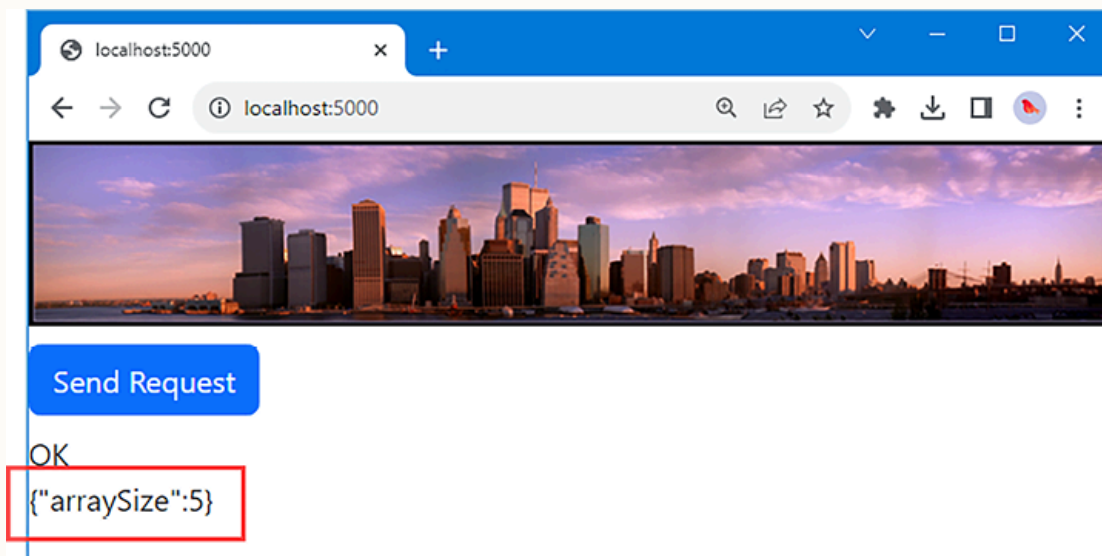


Figure 6.12: Using the Express JSON middleware

Summary

In this chapter, I described the API features that Node.js provides to read and write data, particularly when processing an HTTP request:

- Streams are used as abstract representations of sources and destinations for data, including HTTP requests and responses.
- Data is buffered when it is written to a stream, but it is a good idea to avoid excessive buffering because it can exhaust system resources.
- Data can be read from a stream by handling events or using a `for` loop.
- Data can be piped from a readable stream to a writable stream.
- Data can be transformed as it is piped and can be between JavaScript objects and strings/byte arrays.
- Node.js provides an API to work with files, but third-party packages are the safest way to work with files in a web server project.
- Third-party packages, such as Express, provide enhancements to the Node.js streams to perform common tasks, such as decoding JSON data.

In the next chapter, I describe two aspects of web development in which Node.js works together with other components to deliver an application.