



Data Wrangling with JavaScript

Sample Chapter

Ashley Davis





***Data Wrangling
with JavaScript***

by Ashley Davis

Chapter 12

Copyright 2019 Manning Publications

brief contents

- 1 ■ Getting started: establishing your data pipeline 1
- 2 ■ Getting started with Node.js 25
- 3 ■ Acquisition, storage, and retrieval 59
- 4 ■ Working with unusual data 99
- 5 ■ Exploratory coding 115
- 6 ■ Clean and prepare 143
- 7 ■ Dealing with huge data files 168
- 8 ■ Working with a mountain of data 191
- 9 ■ Practical data analysis 217
- 10 ■ Browser-based visualization 247
- 11 ■ Server-side visualization 274
- 12 ■ Live data 299
- 13 ■ Advanced visualization with D3 329
- 14 ■ Getting to production 358

12

Live data

This chapter covers

- Working with a real-time data feed
- Receiving data through HTTP POST and sockets
- Decoupling modules in your server with an event-based architecture
- Triggering SMS alerts and generating automated reports
- Sending new data to a live chart through socket.io

In this chapter we bring together multiple aspects of data wrangling that we've already learned and combine them into a real-time data pipeline. We're going to build something that's almost a real production system. It's a data pipeline that will do all the usual things: acquire and store data (chapter 3), clean and transform the data (chapter 6), and, in addition, perform on-the-fly data analysis (chapter 9).

Output from the system will take several forms. The most exciting will be a browser-based visualization, based on our work from chapter 10, but with live data feeding in and updating as we watch. It will automatically generate a daily report

(using techniques from chapter 11) that's emailed to interested parties. It will also issue SMS text message alerts about unusual data points arriving in the system. To be sure, the system we'll build now will be something of a toy project, but besides that, it will demonstrate many of the features you'd want to see in a real system of this nature, and on a small scale, it could work in a real production environment.

This will be one of the most complex chapters yet, but please stick with it! I can promise you that getting to the live visualization will be worth it.

12.1 *We need an early warning system*

For many cities, monitoring the air quality is important, and in certain countries, it's even regulated by the government. Air pollution can be a real problem, regardless of how it's caused. In Melbourne, Australia, in 2016, an incident occurred that the media were calling thunderstorm asthma.

A major storm hit the city, and the combination of wind and moisture caused pollen to break up and disperse into particles that were too small to be filtered out by the nose. People with asthma and allergies were at high risk. In the following hours, emergency services were overwhelmed with the large volume of calls. Thousands of people became ill. In the week that followed, nine people died. Some kind of early warning system might have helped prepare the public and the emergency services for the impending crisis, so let's try building something like that.

In this chapter, we'll build an air quality monitoring system. It will be somewhat simplified but would at least be a good starting point for a full production system. We're building an early warning system, and it must raise the alarm as soon as poor air quality is detected.

What are we aiming for here? Our live data pipeline will accept a continuous data feed from a hypothetical air quality sensor. Our system will have three main features:

- To allow air quality to be continuously monitored through a live chart
- To automatically generate a daily report and email it to interested parties
- To continuously check the level of air quality and to raise an SMS text message alert when poor air quality is detected

This chapter is all about dealing with live and dynamic data, and we'll try to do this in a real context. We'll see more software architecture in this chapter than we've yet seen in the book because the work we're doing is getting more complex and we need more powerful ways to organize our code. We'll work toward building our application on an event-based architecture. To emulate how I'd really do the development, we'll start simple and then restructure our code partway through to incorporate an event hub that will decouple the components of our app and help us to manage the rising level of complexity.

12.2 Getting the code and data

The code and data for this chapter are available in the Data Wrangling with JavaScript Chapter 12-repository in GitHub: <https://github.com/data-wrangling-with-javascript/chapter-12>. Data for this chapter was acquired from the Queensland Government open data website at <https://data.qld.gov.au/>.

Each subdirectory in the code repo is a complete working example, and each corresponds to code listings throughout this chapter. Before attempting to run the code in each subdirectory, please be sure to install the npm and Bower dependencies as necessary. Refer to “Getting the code and data” in chapter 2 for help on getting the code and data.

12.3 Dealing with live data

Creating a live data pipeline isn’t much different from anything else we’ve seen so far in the book, except now we’ll have a continuous stream of data pushed to us by a communication channel. Figure 12.1 gives the simplified overall picture. We’ll have an air pollution sensor (our data collection device) that submits the current metric of air quality to our Node.js server on an hourly basis, although we’ll speed this up dramatically for development and testing.

For a more in-depth understanding of how the data feed fits into our pipeline, see figure 12.2. Incoming data arrives in our system on the left of the diagram at the data collection point. The data then feeds through the processing pipeline. You should recognize the various pipeline stages here and already have an idea what they do. Output is then delivered to our user through alerts, visualizations, and a daily report.

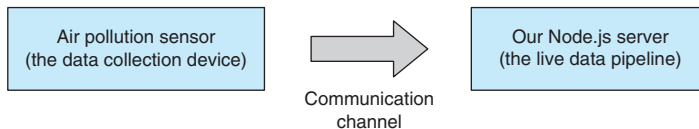


Figure 12.1 An air pollution sensor pushes data to our Node.js server.

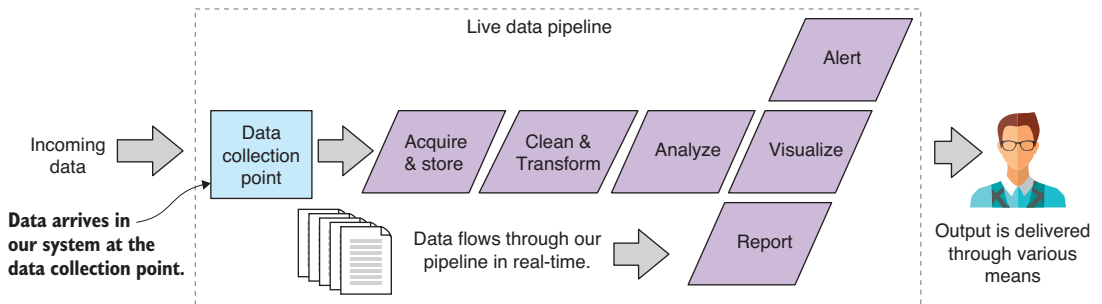


Figure 12.2 We’ll now have a continuous stream of data flowing into our data pipeline.


12.4 Building a system for monitoring air quality

Before we dive into building our air quality monitoring system, let's look at the data we have. The CSV data file *brisbanecbd-aq-2014.csv* is available under the *data* subdirectory of the Chapter-12 GitHub repository. As usual, we should take a good look at our data before we start coding. You can see an extract from the data file in figure 12.3. This data was downloaded from the Queensland Government open data website.¹ Thanks to the Queensland Government for supporting open data.

The data file contains an hourly reading of atmospheric conditions. The metric of interest is the PM10 column. This is the count of particles in the air that are less than 10 micrometers in diameter. Pollen and dust are two examples of such particles. To understand how small this is, you need to know that a human hair is around 100 micrometers wide, so 10 of these particles can be placed on the width of a human hair. That's tiny.

Particulate matter this small can be drawn into the lungs, whereas bigger particles are often trapped in the nose, mouth, and throat. The PM10 value specifies mass per volume, in this case micrograms per cubic meter ($\mu\text{g}/\text{m}^3$).

PM10 is the column we are interested in.



	A	B		G	H
1	Date	Wind Direc		PM10 ($\mu\text{g}/\text{m}^3$)	Bsp (Mm^{-1})
2	1/01/2014 0:00	154		23.2	21.6
3	1/01/2014 1:00	151		22.4	19.3
4	1/01/2014 2:00	13		21.7	18.9
5	1/01/2014 3:00	14		21.3	21.6
6	1/01/2014 4:00	11		19.9	20.1
7	1/01/2014 5:00	148		16.6	18.2
8	1/01/2014 6:00	171		4.8	18.5
9	1/01/2014 7:00	151		15.9	10.9
10	1/01/2014 8:00	14	...	27.9	11
11	1/01/2014 9:00	13		29.6	11.7
12	1/01/2014 10:00	11		31	10.5
13	1/01/2014 11:00	84		17.3	18.8
14	1/01/2014 12:00	77		17.7	12.6
15	1/01/2014 13:00	47		0.2	136.7
16	1/01/2014 14:00	57		0.3	147.1
17	1/01/2014 15:00	31		29	43.5
18	1/01/2014 16:00	3		26.4	27.6
19	1/01/2014 17:00	5		25	15.2
20	1/01/2014 18:00	1		23.8	15.7

These large values (greater than 80) indicate poor air quality at this time.




Figure 12.3 The data for this chapter. We're interested in the PM10 column for monitoring air quality.

¹ For more information, see <https://data.qld.gov.au/dataset/air-quality-monitoring-2014>.

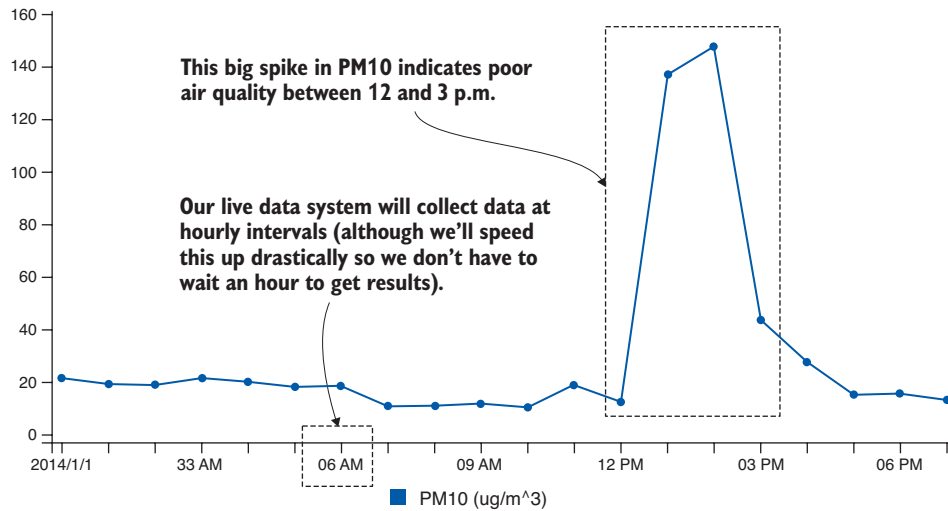


Figure 12.4 Charting the PM10 value, we can see the big spike between 12 and 3 p.m.

Notice the larger values for PM10 that are highlighted in figure 12.3. At these times, we've got potentially problematic levels of atmospheric particulate matter. On the chart in figure 12.4, we can easily see this spike between 12 p.m. and 3 p.m.—this is when air quality is worse than normal. Figure 12.4 also shows the chart that we'll make in this chapter.

For the purposes of our air quality monitoring system, we'll regard any PM10 value over 80 as a poor quality of air and worthy of raising an alarm. I've taken this number from the table of air quality categories from the Environmental Protection Authority (EPA) Victoria.

What will our system look like? You can see a schematic of the complete system in figure 12.5. I'm showing you this system diagram now as a heads-up on where we're heading. I don't expect you to understand all the parts of this system right at the moment, but you can think of this as a map of what we're creating, and please refer back to it from time to time during this chapter to orient yourself.

I told you this would be the most complicated project in the book! Still, this system will be simple compared to most real production systems. But it will have all the parts shown in the schematic even though we'll only be examining parts of this whole. At the end of the chapter, I'll present the code for the completed system for you to study in your own time.

Our system starts with data produced by an air pollution sensor (shown on the left of figure 12.5). The sensor detects the air quality and feeds data to the data collection point at hourly intervals. The first thing we must do is store the data in our database. The worst thing we can do is lose data, so it's important to first make sure the data is safe. The data collection point then raises the incoming-data event. This is where our event-based architecture comes into play. It allows us to create a separation of concerns and decouple our data collection from our downstream data operations. To the right of figure 12.5, we see the outputs of our system, the SMS alert, the daily report, and the live visualization.

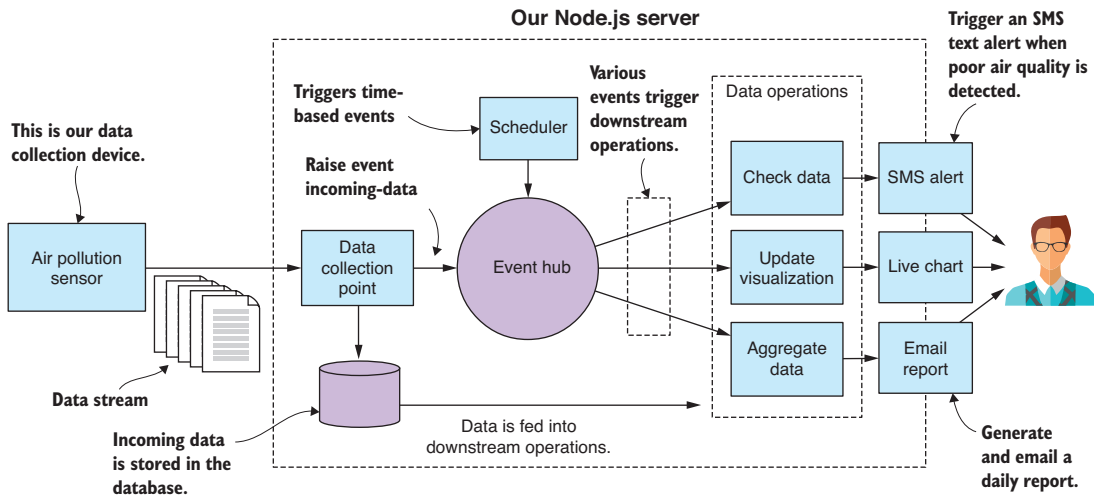


Figure 12.5 Schematic of our air quality monitoring system

12.5 Set up for development

To build this system, we must create a kind of artificial scaffold in which to run it. You probably don't have an actual particulate matter sensor on hand—although you can actually buy these at a reasonable price if you're particularly motivated by this example project.

Instead, we'll use JavaScript to create a sort of mock sensor to simulate the real sensor. The code we'll write might be pretty close to what the real thing would look like. For example, if we could attach a Raspberry PI to the real sensor and install Node.js, we could then run code that might be similar to the mock sensor we're going to build in a moment.

We don't have a real sensor, so we'll need precanned data for the mock sensor to “generate” and feed to our monitoring system. We already have realistic data, as seen in figure 12.3, although this data is hourly. If we're to use it in a realistic fashion, then our workflow would be slow because we'd have to wait an hour for each new data point to come in.

To be productive, we need to speed this up. Instead of having our data come in at hourly intervals, we'll make it come in every second. This is like speeding up time and watching our system run in fast forward. Other than this time manipulation, our system will run in a realistic fashion.

Each code listing for this chapter has its own subdirectory under the Chapter-12 GitHub repository. Under each listing's directory, you'll find a client and a server directory. You can get an idea of what this looks like in figure 12.6.

For each code listing, the mock sensor, our data collection device, lives in the client subdirectory, and our evolving air monitoring system lives in the server subdirectory. To follow along with the code listings, you'll need to open two command-line windows. In the first command line, you should run the server as follows:

```
cd listing-12.1
cd server
node index.js
```

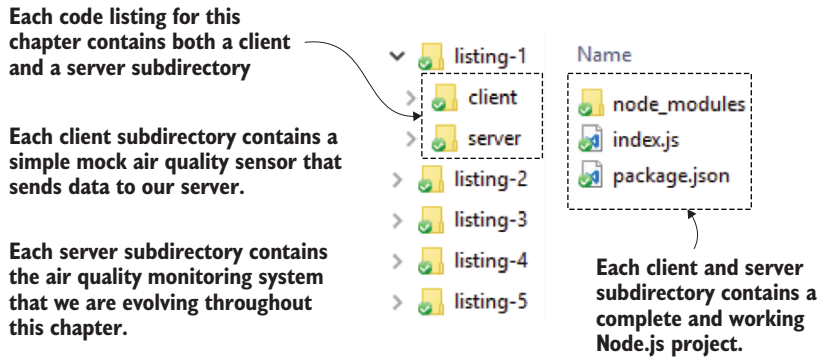


Figure 12.6 The project structure for code listings in chapter 12

In the second command line, you should run the client (mock sensor) as follows:

```
cd listing-12.1
cd client
node index.js
```

The client and server are now both running, and the client is feeding data to the server. When moving onto the next code listing, change the listing number depending on where you are. Make sure you install the npm and Bower dependencies before trying to run each code listing.

Live reload

Don't forget that you can also use `nodemon` in place of `node` when running scripts to enable live reload, which allows you to make changes to the code. `nodemon` will automatically rerun your code without you having to restart it manually. Please check chapter 5 for a refresher on this.

12.6 Live-streaming data

The first problem we must solve is how to connect our sensor to our monitoring system. In the coming sections, we'll cover two network-based mechanisms: HTTP POST and sockets. Both protocols build on the TCP network protocol and are directly supported by Node.js. Which protocol you choose depends on the frequency at which you expect data to be submitted.

12.6.1 HTTP POST for infrequent data submission

Let's start by looking at data submission via HTTP POST. We can use this when data submission is infrequent or ad hoc. It's also simplest and so is a good place to start. Figure 12.7 shows how our air pollution sensor is going to send single packets of data to our Node.js server. In this case, our data collection point, the entry point for data arriving at our server, will be an HTTP POST request handler. From there, the data is fed into our live data pipeline.

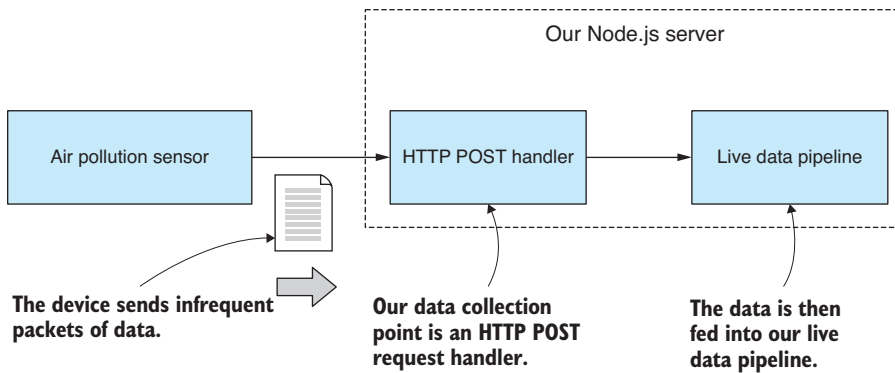


Figure 12.7 HTTP POST is used to send single packets of data to our server.

Our code at this point will be incredibly simple. Starting off, we want to get the data feed moving from the mock sensor into our Node.js server. You can run this code, but you must start it in the right order—first the server and then the client (mock sensor). Our Node.js server receives data and then prints it to the console (as shown in figure 12.8). We’re starting simple, and that’s all it does at this point. We do this to check that our data is coming across to our server correctly.

Node.js directly supports HTTP POST, but in this case, we’ll use *request-promise*, a higher-level library, to make this a bit easier and also to wrap our HTTP request in promises.

If you installed dependencies already, then you have *request-promise* installed in your project; otherwise, you can install it in a fresh Node.js project like this:

```
npm install --save request-promise
```

```
> node index.js
Data collection point listening on port 3000!
{ Date: '01/01/2014 00:00',
  'Wind Direction (degTN)': 154,
  'Wind Speed (m/s)': 1.1,
  'Wind Sigma Theta (deg)': 44.5,
  'Wind Speed Std Dev (m/s)': 0.5,
  'Air Temperature (degC)': 23.2,
  'PM10 (ug/m^3)': 21.6,
  'Bsp (Mm^-1)': '',
  Location: 'brisbanecbd' }
{ Date: '01/01/2014 01:00',
  'Wind Direction (degTN)': 153,
  'Wind Speed (m/s)': 1.3,
  'Wind Sigma Theta (deg)': 39.1,
  'Wind Speed Std Dev (m/s)': 0.7,
  'Air Temperature (degC)': 22.4,
  'PM10 (ug/m^3)': 19.3,
  'Bsp (Mm^-1)': 27,
  Location: 'brisbanecbd' }
```

Figure 12.8 Output displayed as our Node.js server receives data using HTTP POST.

The following listing shows the code for our first mock air pollution sensor. It reads our example CSV data file. Once per second it takes the next row of data and submits it to the server using HTTP POST.

Listing 12.1a Air pollution sensor that submits data to the server via HTTP POST (listing-12.1/client/index.js)

```

const fs = require('fs');
const request = require('request-promise');
const importCsvFile = require('./toolkit/importCsvFile.js');

const dataFilePath = "../data/brisbane cbd-aq-2014.csv";
const dataSubmitUrl = "http://localhost:3000/data-collection-point";

importCsvFile(dataFilePath)
  .then(data => {
    let curIndex = 0;

    setInterval(() => {
      const outgoingData = Object.assign({}, data[curIndex]);
      curIndex += 1;

      request.post({
        uri: dataSubmitUrl,
        body: outgoingData,
        json: true
      }, 1000);
    }, 1000);
  })
  .catch(err => {
    console.error("An error occurred.");
    console.error(err);
  });

```

Loads the example data from the CSV file (points to `importCsvFile`)

This is the path to the CSV file containing example data. (points to `dataFilePath`)

This is the URL for submitting data to our Node.js server. (points to `dataSubmitUrl`)

Clones the data so we can modify it without overwriting the original (points to `Object.assign({}, data[curIndex])`)

Once per second, it sends a chunk of data to the server. (points to `setInterval`)

Iterates through the example data one row at a time (points to the `setInterval` loop)

Uses HTTP POST to submit a packet of data to the server (points to `request.post`)

Specifies the URL to submit data to (points to `uri: dataSubmitUrl`)

This is the data being submitted. (points to `body: outgoingData`)

Uses JSON encoding. The data is sent over the wire using the JSON data format. (points to `json: true`)

On the server side, we use the *express* library to accept incoming data using HTTP POST. As we did with *request-promise*, we use the *express* library to make our lives a little easier. Node.js already has everything we need to build an HTTP server, but it's common practice to use a higher-level library like *express* to simplify and streamline our code.

Again, if you installed dependencies, then you already have the *express* library installed; otherwise, you install it and the *body-parser* middleware as follows:

```
npm install --save express body-parser
```

We're using the *body-parser* middleware to parse the HTTP request body from JSON when it's received. This way we don't have to do the parsing ourselves. It will happen automatically.

Listing 12.1b shows the code for a simple Node.js server that accepts data using the URL `data-collection-point`. We print incoming data to the console to check that it's coming through correctly.

Listing 12.1b Node.js server that can accept data via HTTP POST (listing-12.1/server/index.js)

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');

app.use(bodyParser.json());

app.post("/data-collection-point", (req, res) => {
  console.log(req.body);
  res.sendStatus(200);
});

app.listen(3000, () => { // Start the server.
  console.log("Data collection point listening on port 3000!");
});
```

Requires the body-parser middleware so that the HTTP request body is automatically parsed from JSON data

Defines a REST API endpoint that receives packets of data that were submitted to the server

We're not doing anything with the data yet, only printing to check that it's coming through.

Responds to the client with HTTP status 200 (status okay)

We now have a mechanism that allows us to accept an infrequent or ad hoc data feed. This would be good enough if we were only receiving incoming data on an hourly basis—as we would be if this were a real-life system. But given that we're sending our data through every second, and because it's an excuse to do more network coding, let's look at using sockets to accept a high-frequency real-time data feed into our server.

12.6.2 Sockets for high-frequency data submission

We'll now convert our code over to using a socket connection, which is a better alternative when we have a high frequency of data submission. We're going to create a long-lived communication channel between the sensor and the server. The communication channel is also bidirectional, but that's not something we'll use in this example, although you could later use it for sending commands and status back to your sensor if that's what your system design needed.

Figure 12.9 shows how we'll integrate the socket connection into our system. This looks similar to what we did with HTTP POST, although it shows that we'll have a stream

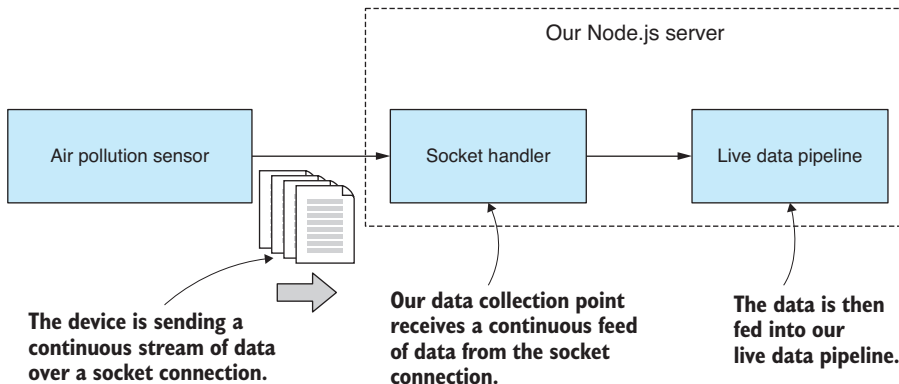


Figure 12.9 A long-lived socket connection is used to receive continuous and high-frequency streaming data into our server.

of data coming through and arriving at the socket handler, which replaces the HTTP post handler and is our new data collection point.

In the following listing, we adapt our mock sensor from listing 12.1a so that it writes the outgoing data to the socket connection. Besides the connection setup and the call to `socket.write`, this listing is similar to listing 12.1a.

Listing 12.2a Air pollution sensor that submits data to the server via a socket connection (listing-12.2/client/index.js)

```
// ... initial setup as per listing 12.1a ...

const serverHostName = "localhost";
const serverPortNo = 3030;

const client = new net.Socket();
client.connect(serverPortNo, serverHostName, () => {
  console.log("Connected to server!");
});

client.on("close", () => {
  console.log("Server closed the connection.");
});

importCsvFile(dataFilePath)
  .then(data => {
    let curIndex = 0;

    setInterval(() => {
      const outgoingData = Object.assign({}, data[curIndex]);
      curIndex += 1;

      const outgoingJsonData = JSON.stringify(outgoingData);

      client.write(outgoingJsonData);

    }, 1000);
  })
  .catch(err => {
    console.error("An error occurred.");
    console.error(err);
  });
```

Sets up the server connection details

Connects the socket to our Node.js server

This callback is invoked when the server has closed the connection.

Loads the example data from the CSV file

Once per second, it sends a chunk of data to the server.

Serializes outgoing data to JSON format

Sends JSON data over the wire

In listing 12.2b, we have a new Node.js server that listens on a network port and accepts incoming socket connections. When our mock sensor (the client) connects, we set a handler for the socket's data event. This is how we intercept incoming data; we're also starting to see that event-based architecture that I mentioned earlier. In this example, as before, we print the data to the console to check that it has come through correctly.

Listing 12.2b Acquiring real-time data through a socket connection (listing-12.2 /server/index.js)

```

const net = require('net');

const serverHostName = "localhost";
const serverPortNo = 3030;

const server = net.createServer(socket => {
  console.log("Client connected!");

  socket.on("data", incomingJsonData => {
    const incomingData = JSON.parse(incomingJsonData);

    console.log("Received: ");
    console.log(incomingData);

    socket.on("close", () => {
      console.log("Client closed the connection");
    });

    socket.on("error", err => {
      console.error("Caught socket error from client.");
      console.error(err);
    });
  });

  server.listen(serverPortNo, serverHostName, () => {
    console.log("Waiting for clients to connect.");
  });
});

```

Deserializes incoming JSON data →

This callback is invoked when the client has closed the connection. →

Sets up the server connection details

Creates the socket server for data collection

Handles incoming-data packets

Logs data received so that we can check that it's coming through okay

Adds an error handler, mainly for ECONNRESET when the client abruptly disconnects

Starts listening for incoming socket connections

Note how we're sending the data over the wire in the JSON data format. We did this in the HTTP example as well, but in that case request-promise (on the client) and express (on the server) did the heavy lifting for us. In this case, we're manually serializing the data to JSON (on the client) before pushing it onto the network and then manually deserializing when it comes out at the other end (on the server).

12.7 Refactor for configuration

To this point, our server code has been simple, but in a moment the complexity will start to rise sharply. Let's take a moment and do a refactor that will cleanly separate our configuration from our code. We won't go too far with this; it's a simple restructure and will help us keep the app tidy as it grows.

The only configuration we have at the moment is the socket server setup details from listing 12.2b. We're going to move these to a separate configuration file, as shown in figure 12.10. This will be a central place to consolidate the configuration of the app and where we'll need to go to later change its configuration.

Listing 12.3a shows our simple starting configuration for the project. You might well ask, "Why bother?" We'll, it's because we have a bunch of configuration details yet to

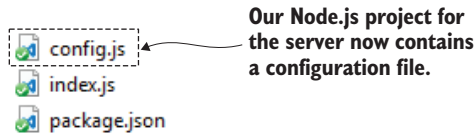


Figure 12.10 The new configuration file in our Node.js project

come. The database, SMS alerts, and report generation all require their own configuration, and it's nice to gather them in this one place.

Listing 12.3a Adding a simple configuration file to the Node.js project (listing-12.3/server/config.js)

```
module.exports = {
  server: {
    hostName: "localhost",
    portNo: 3030
  }
};
```

— The first details in our configuration file; specifies the server configuration

Listing 12.3b shows how we load and use the configuration file. Nothing is complex here; our configuration is a regular Node.js code module with exported variables. This is a simple and convenient way to get started adding configuration to your app. It costs us little time to get this in place, and it's useful in the long run.

Listing 12.3b The Node.js server is modified to load and use the configuration file (listing-12.3/server/index.js)

```
const net = require('net');
const config = require('./config.js');

const server = net.createServer(socket => {
  // ... code omitted, same as listing 12.1b ...
});

server.listen(config.server.portNo, config.server.hostName, () => {
  console.log("Waiting for clients to connect.");
});
```

← Loads the configuration file just like any other Node.js code module

Starts the socket server with details loaded from the configuration file

You may wonder why I chose to use a Node.js code module as a configuration file. Well, my first thought was for simplicity. Normally, in production, I've used a JSON file for this kind of thing, and that's just as easy to drop into this example. Believe it or not, you can require a JSON file in Node.js the same way that you require a JavaScript file. For example, you could have also done this:

```
const config = require('./config.json');
```

It's cool that you can do that: it's a simple and effective way to load data and configuration into your Node.js app. But it also occurred to me that using JavaScript as your configuration file means you can include comments! This is a great way to document and explain configuration files and isn't something you can ordinarily do with JSON files. (How many times do you wish you could have added comments to JSON files?!)

You have more scalable and secure ways to store configuration, but simplicity serves our needs here, and this is something we'll touch on again in chapter 14.

12.8 Data capture

Now we're more than ready to do something with our data, and the first thing we should do is to make sure that it's safe and secure. We should immediately capture it to our database so that we're at no risk of losing it.

Figure 12.11 shows what our system looks like at this point. We have data incoming from the sensor, the data arrives at the data collection point, and then it's stored in our database for safe-keeping. This time, after we run our code, we use a database viewer such as Robomongo to check that our data has arrived safely in our database (see figure 12.12).

To connect to the database, we need to get our database connection details from somewhere. In the following listing, we've added these to our configuration file.

Listing 12.4a Adding the database connection details to the configuration file (listing-12.4/server/config.js)

```
module.exports = {
  server: {
    hostname: "localhost",
    portNo: 3030
  },

  database: {
    host: "mongodb://localhost:27017",
    name: "air_quality"
  }
};
```

These are the connection details for our database.

Note that we're using the default port 27017 when connecting to MongoDB in listing 12.4a. This assumes that you have a default installation of MongoDB on your development PC. If you want to try running this code, you'll need to install MongoDB; otherwise, you could boot up the Vagrant VM that's in the `vm-with-empty-db` subdirectory

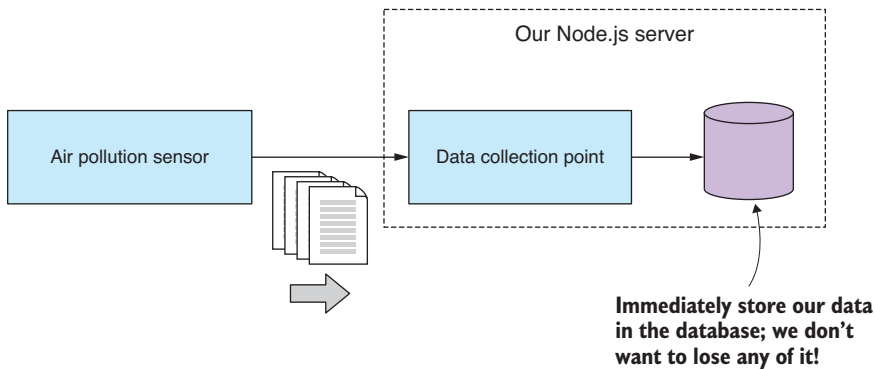
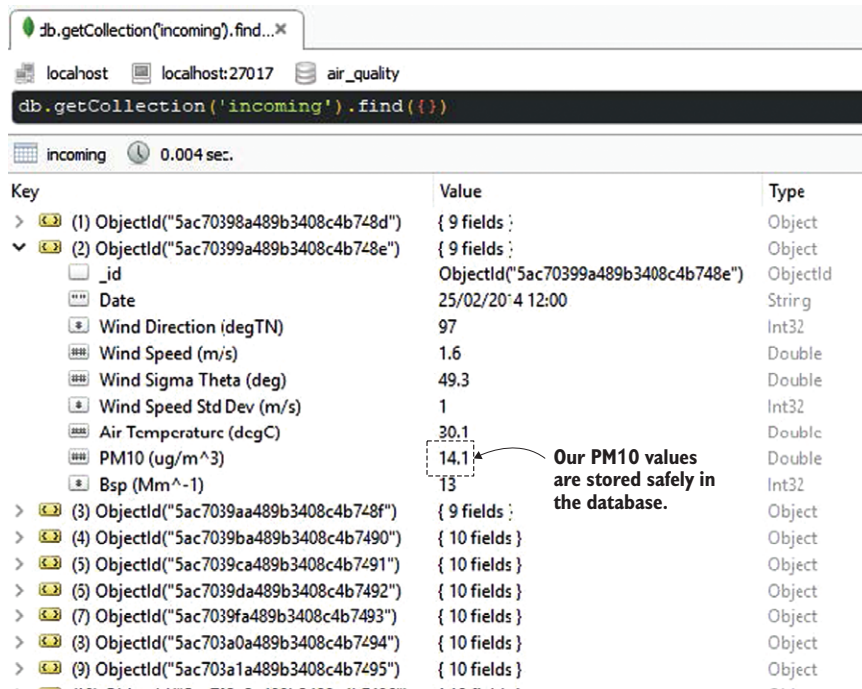


Figure 12.11 Immediately store received data into our database before taking any further action.



db.getCollection('incoming').find({})

incoming 0.004 sec.

Key	Value	Type
> (1) ObjectId("5ac70398a489b3408c4b748d")	{ 9 fields }	Object
▼ (2) ObjectId("5ac70399a489b3408c4b748e")	{ 9 fields }	Object
_id	ObjectId("5ac70399a489b3408c4b748e")	ObjectId
Date	25/02/2014 12:00	String
Wind Direction (degTN)	97	Int32
Wind Speed (m/s)	1.6	Double
Wind Sigma Theta (deg)	49.3	Double
Wind Speed Std Dev (m/s)	1	Int32
Air Temperature (degC)	30.1	Double
PM10 (ug/m^3)	14.1	Double
Bsp (Mm^-1)	13	Int32
> (3) ObjectId("5ac7039aa489b3408c4b748f")	{ 9 fields }	Object
> (4) ObjectId("5ac7039ba489b3408c4b7490")	{ 10 fields }	Object
> (5) ObjectId("5ac7039ca489b3408c4b7491")	{ 10 fields }	Object
> (6) ObjectId("5ac7039da489b3408c4b7492")	{ 10 fields }	Object
> (7) ObjectId("5ac7039fa489b3408c4b7493")	{ 10 fields }	Object
> (8) ObjectId("5ac703a0a489b3408c4b7494")	{ 10 fields }	Object
> (9) ObjectId("5ac703a1a489b3408c4b7495")	{ 10 fields }	Object

Our PM10 values are stored safely in the database.

Figure 12.12 Using Robomongo to check that our incoming data has been captured to the database

of the Chapter-8 Github repository. Booting that VM will give you an empty MongoDB database on port 6000 to use for code listings in this chapter. Make sure you modify the code to refer to the correct port number. For example, in listing 12.4a you'd change the connection string from `mongodb://localhost:27017` to `mongodb://localhost:6000`. For help on Vagrant, please see appendix C.

The following listing shows the code that connects to MongoDB and stores the data that arrives at our data collection point immediately after it's received.

Listing 12.4b Storing incoming data into the MongoDB database (listing-12.4 /server/index.js)

```
const mongodb = require('mongodb');
const net = require('net');
const config = require('./config.js');

mongodb.MongoClient.connect(config.database.host)
  .then(client => {
    const db = client.db(config.database.name);
    const collection = db.collection("incoming");

    console.log("Connected to db");

    const server = net.createServer(socket => {
      console.log("Client connected!");
    });
```

Retrieves the MongoDB collection where we'll store incoming data

Opens a connection to the database server before we start accepting incoming data

Retrieves the database we're using

```

socket.on("data", incomingJsonData => {
  console.log("Storing data to database.");

  const incomingData = JSON.parse(incomingJsonData);

  collection.insertOne(incomingData)
    .then(doc => {
      console.log("Data was inserted.");
    })
    .catch(err => {
      console.error("Error inserting data.");
      console.error(err);
    });

  socket.on("close", () => {
    console.log('Client closed the connection');
  });

  socket.on("error", err => {
    console.error("Caught socket error from client.");
    console.error(err);
  });
});

server.listen(config.server.portNo, config.server.hostName, () => {
  console.log("Waiting for clients to connect.");
});

```

The data was inserted successfully.

Shows that something went wrong while inserting the data

Inserts incoming data into the database

The fact that we're storing this data in the database immediately after receiving it is a design decision. I believe that this data is important and that we shouldn't risk doing any initial processing on it before we've safely stored it. We'll touch on this idea again soon.

12.9 An event-based architecture

Let's now look at how we can better evolve our application over time. I wanted an opportunity to show how we can deploy a design pattern to structure our app and help manage its complexity.

You might argue that I'm overengineering this simple toy application, but what I want to show you is how separation of concerns and decoupling of components can give us the foundation for a solid, reliable, and extensible application. This should become obvious as we ramp up complexity culminating in the complete system at the end of the chapter.

Figure 12.13 shows how we'll use an event hub to decouple our data collection from any downstream data processing operation; for example, *update visualization*, which is responsible for forwarding incoming data to a live chart in the web browser.

The event hub is like a conduit for our events: the incoming-data event is raised by the data collection point, and the update visualization event handler responds to it. With this kind of infrastructure in place, we can now easily slot in new downstream data operations to extend the system.

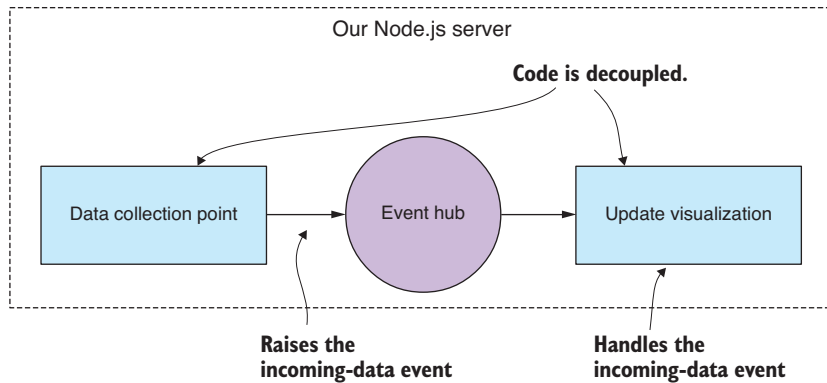


Figure 12.13 An event-handling architecture allows us to decouple our code modules.

Figure 12.14, for example, shows how we'll plug in an SMS alert module so that our system can raise the alarm when it has detected poor-quality air.

Using an event-based architecture like this gives us a framework on which to hang new code modules. We've added a natural extension point where we can plug in new event sources and event handlers. This means we've designed our application to be upgraded. We're now better able to modify and extend our app without turning it into a big mess of spaghetti code—at least that's the aim. I won't claim that it's easy to keep an evolving application under control, but design patterns like this can help.

The important thing for us in this project is that we can add new code modules such as *update visualization* and *SMS alert* without having to modify our data collection point. Why is this important here and now? Well, I wanted to make the point that the safety of our data is critical, and we must ensure that it's safe and sound before anything else

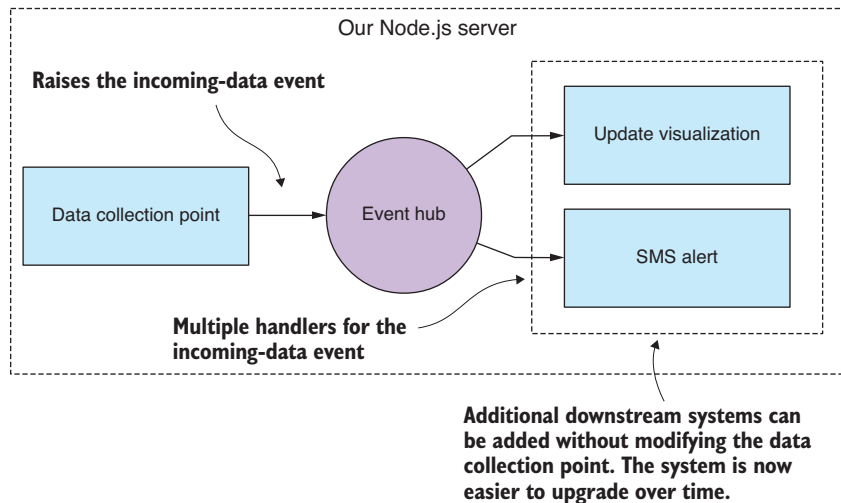


Figure 12.14 We can now expand our system, adding new downstream operations without refactoring or restructuring the data collection point.

happens. Any time we make code changes to the data collection point, we run the risk of breaking this code. It's imperative that we minimize the changes that we make to this code in the future, and the event-based architecture means we can add new code modules without having to change the code at the data collection point.

As well as helping structure our app and make it more extensible, the event-based architecture also makes it easy to partition our system so that, if necessary for scaling up, we can distribute the application across multiple servers or virtual machines with the events being transmitted across the wire. This kind of architecture can help enable horizontal scaling that we'll discuss further in chapter 14.

12.10 Code restructure for event handling

Let's restructure our code so that it's based around the notion of an event hub that coordinates the raising and handling of events. We'll use the Node.js `EventEmitter` class because it's designed for this sort of thing.

In listing 12.5a you can see the code for our new event hub. This is super simple: the entire module instantiates an `EventEmitter` and exports it for use in other modules. No one said this needed to be complex, although you can surely build a more sophisticated event hub than this!

Listing 12.5a Creating an event hub for the server (listing-12.5/server/event-hub.js)

```
const events = require('events');
const eventHub = new events.EventEmitter();
module.exports = eventHub;
```

Instantiates a Node.js `EventEmitter` as our event hub

Exports the event hub for other modules to rely on

Now that we have our event hub, we can wire it up to the existing code. The first thing we have to do is raise the incoming-data event when data is received by the server. We do this by calling the `emit` function on the event hub.

As you can see from the code extract in the following listing, the event is raised immediately after the data has been successfully stored in the database. For safety, we store the data first and everything else happens later.

Listing 12.5b Raising the incoming-data event (extract from listing-12.5/server/data-collection-point.js)

```
incomingDataCollection.insertOne(incomingData)
  .then(doc => {
    eventHub.emit('incoming-data', incomingData);
  })
  .catch(err => {
    console.error("Error inserting data.");
    console.error(err);
  });
```

Inserts data into the database

Raises the incoming-data event and passes through the data

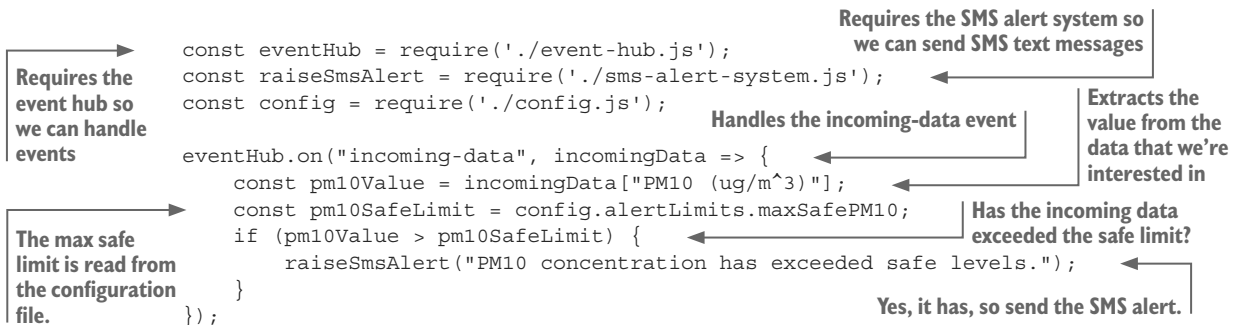
With the incoming-data event in place and being raised whenever we have data arriving at the server, we're in a position to start building downstream data processing modules.

12.10.1 Triggering SMS alerts

The next thing we care about is knowing in real time when the quality of the air is deteriorating. We can now add an event handler to monitor incoming PM10 values and raise an alarm when poor air quality is detected.

To handle the event, we first import the event hub into our code. Then we call the `on` function to register an event handler function for a named event such as the `incoming-data` event we added a moment ago. This is shown in the following listing: checking the incoming data for PM10 values greater than or equal to the max safe level, which is set to 80 in the configuration file. When such values are detected, we sound the alarm and send an SMS text message to our users.

Listing 12.5c Handle event and trigger alert when PM10 exceeds safe value (listing-12.5/server/trigger-sms-alert.js)



The code in listing 12.5c is an example of adding a downstream data operation that does data analysis and sequences an appropriate response. This code is simple, but we could imagine doing something more complex here, such as checking whether the rolling average (see chapter 9) is on an upward trend or whether the incoming value is more than two standard deviations above the normal average (again, see chapter 9). If you'd prototyped data analysis code using exploratory coding (such as we did in chapter 5 or 9), you can probably imagine slotting that code into the system at this point.

Now if you run this code (listing 12.5) and wait for a bit, you'll see an "SMS alert" triggered. You only have to wait a few moments for this to happen (when those large PM10 values between 12 p.m. and 3 p.m. come through). The code that would send the SMS message is commented out for the moment, though, so all you'll see is console logging that shows you what would have happened.

To get the SMS code working, you'll need to uncomment the code in the file listing-12.5/server/sms-alert-system.js. You'll need to sign up for Twilio (or similar service) and add your configuration details to the config file. Also make sure you add your own mobile number so that the SMS message will be sent to you. Do all this, run the code again, and you'll receive the alert on your phone.

12.10.2 Automatically generating a daily report

Let's look at another example of raising and handling events. For the next feature, we'll add automatically generated daily reports. The report won't be anything fancy; we'll render a chart of PM10 to a PDF file and then have that emailed to our users. But you can imagine going much further with this, say, rendering other statistics or attaching a spreadsheet with a summary of recent data.

Because we want to generate our reports daily, we now need a way to generate time-based events. For this, we'll add a scheduler to our system, and we'll program it to raise a *generate-daily-report* event once per day. A separate daily report generator module will handle the event and do the work. You can see how this fits together in figure 12.15.

To implement the scheduler, we'll need a timer to know when to raise the event. We could build this from scratch using the JavaScript functions `setTimeout` or `setInterval`. Although these functions are useful, they're also low-level, and I'd like us to use something more expressive and more convenient.

RAISING THE GENERATE DAILY REPORT EVENT

To schedule our time-based events, we'll rely on the `cron` library from npm to be our timer. With this library we can express scheduled jobs using the well-known UNIX cron format. As with any such library, you have many alternatives available on npm; this is the one that I use, but it's always good to shop around to make sure you're working with a library that best suits your own needs.

In listing 12.6a we create an instance of `CronJob` with a schedule retrieved from our config file and then start the job. This invokes `generateReport` once per day, and this is where we raise the *generate-daily-report* event.

Listing 12.6a Using the `cron` library to emit the time-based *generate-daily-report* event (listing-12.6/server/scheduler.js)

```

const eventHub = require('./event-hub.js');
const cron = require('cron');
const config = require('./config.js');

function generateReport () {
  eventHub.emit("generate-daily-report");
};

const cronJob = new cron.CronJob({
  cronTime: config.dailyReport.schedule,
  onTick: generateReport
});

cronJob.start();
  
```

This callback is invoked on a daily schedule. (points to `generateReport` function)

Requires the event hub so we can raise events (points to `require('./event-hub.js')`)

Requires the cron library for scheduled time-based tasks (points to `require('cron')`)

Raises the event *generate-daily-report* and lets the rest of the system deal with it (points to `eventHub.emit("generate-daily-report")`)

Creates the cron job (points to `new cron.CronJob`)

Configures the regular schedule at which to tick the job (points to `config.dailyReport.schedule`)

Specifies the callback to invoke for each scheduled tick (points to `generateReport` in `onTick`)

Starts the cron job (points to `cronJob.start()`)

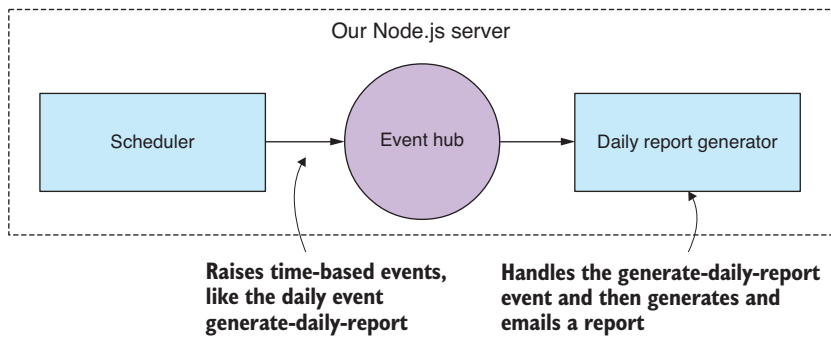


Figure 12.15 Our scheduler feeds an event into the system once per day to generate a daily report.

The cron format we'll use for our daily cron job is specified in the configuration file and looks like this:

```
00 00 06 * * 1-5
```

This looks cryptic, but we can read it from right to left as Monday to Friday (days 1–5), every month (the asterisk), every day of the month (the next asterisk), 6 a.m. at the zero minute, and the zero second. This specifies the time at which to invoke the job. To put it more succinctly: we generate our report each weekday at 6 a.m.

The problem with this schedule is that it takes far too long to test. We can't wait a whole day to test the next iteration of our report generation code! As we did with the incoming-data stream, we need to speed things up, so we'll comment out the daily schedule (we'll need it again to put this app into production) and replace it with one that runs more frequently:

```
00 * * * * *
```

This specifies a schedule that runs every minute (you can read it right to left as every day, every month, every day of month, every hour, every minute, and at the zero second of that minute).

We'll generate a new report every minute. This is a fast pace to be sure, but it means we have frequent opportunities to test and debug our code.

HANDLING THE GENERATE REPORT EVENT

Now we're ready to handle the *generate-daily-report* event and generate and email the report. The following listing shows how the event is handled and then calls down to a helper function to do the work.

Listing 12.6b Handling the generate-daily-report event and generating the report (listing-12.6/server/trigger-daily-report.js)

```
const eventHub = require('./event-hub.js');
const generateDailyReport = require('./generate-daily-report.js');

function initGenerateDailyReport (db) {
```

Requires the event hub so we can handle events

This function initializes our report generation event handler (the database is passed in).

```

eventHub.on("generate-daily-report", () => {
  generateDailyReport(db)
    .then(() => {
      console.log("Report was generated.");
    })
    .catch(err => {
      console.error("Failed to generate report.");
      console.error(err);
    });
});

module.exports = initGenerateDailyReport;

```

Generates the report →

← **Handles the generate-daily-report event**

GENERATING THE REPORT

Generating the report is similar to what we learned in chapter 11; in fact, listing 12.6c was derived from listing 11.7 in chapter 11.

Before generating the report, we query the database and retrieve the data that's to be included in it. We then use the `generateReport` toolkit function, which, the way we did in chapter 11, starts an embedded web server with a template report and captures the report to a PDF file using Nightmare. Ultimately, we call our helper function `sendEmail` to email the report to our users.

Listing 12.6c Generating the daily report and emailing it to interested parties (listing-12.6/server/generate-daily-report.js)

```

const generateReport = require('./toolkit/generate-report.js');
const sendEmail = require('./send-email.js');
const config = require('./config.js');

function generateDailyReport (db) {

  const incomingDataCollection = db.collection("incoming");

  const reportFilePath = "./output/daily-report.pdf";

  return incomingDataCollection.find()
    .sort({ _id: -1 })
    .limit(24)
    .toArray()
    .then(data => {
      const chartData = {
        xFormat: "%d/%m/%Y %H:%M",
        json: data.reverse(),
        keys: {
          x: "Date",
          value: [
            "PM10 (ug/m^3)"
          ]
        }
      };

      return generateReport(chartData, reportFilePath);
    });
}

```

Requires the toolkit function to generate the report →

← **Requires the helper function to send the email**

← **This is a helper function to generate the daily report and email it to interested parties.**

← **This is the file path for the report we're generating and writing to a file.**

Limits to entries for the most recent 24 hours →

← **Queries the database for records**

← **Gets the most recent records first, a convenient method of sorting based on MongoDB ObjectIds**

← **Specifies the format of the Date column used by C3 to parse the data series for the X axis**

← **Reverses the data so it's in chronological order for display in the chart**

Prepares the data to display in the chart →

← **Renders a report to a PDF file**

```

        .then(() => {
            const subject = "Daily report";
            const text = "Your daily report is attached.";
            const html = text;
            const attachments = [
                {
                    path: reportFilePath,
                }
            ];
            return sendEmail(
                config.dailyReport.recipients,
                subject, text, html, attachments
            );
        });
    };

module.exports = generateDailyReport;

```

Specifies attachments to send with the email →

Specifies the subject and the body of the email ←

This could also include a fancy HTML-formatted version of the email here. ←

Attaches our generated report to the email →

We only need a single attachment here, but you could easily add more. ←

Emails the report to specified recipients ←

To run the code for listing 12.6, you'll need to have an SMTP email server that you can use to send the emails. Typically, I'd use Mailgun for this (which has a free/trial version), but you have plenty of other alternatives, such as Gmail. You need access to a standard SMTP account and then can put your SMTP username and password and report-related details in the config file. You can now run listing 12.6 and have it email you a daily report once every minute (please don't leave it running for too long—you'll get a lot of emails!).

You might now be interested to peruse the code in listing-12.6/server/send-email.js to understand how the email is sent using the Nodemailer library (the preeminent Node.js email sending library).

12.11 Live data processing

We'll get to the live visualization in a moment and finish up this chapter, but before that, I want to have a quick word about adding more data processing steps to your live pipeline.

Say that you need to add more code to do data cleanup, transformation, or maybe data analysis. Where's the best place to put this?

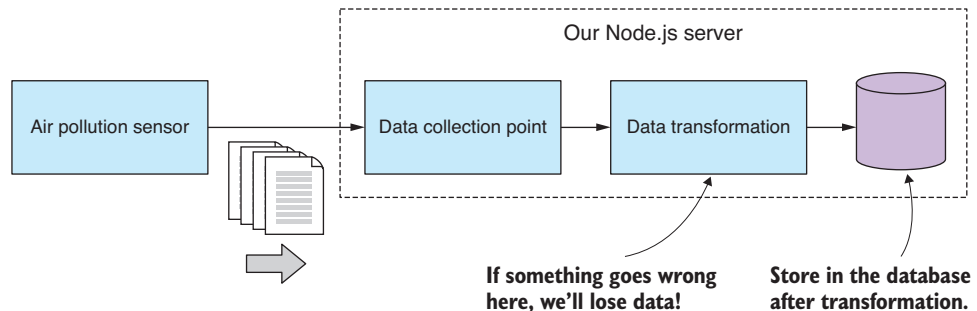


Figure 12.16 Data transformation during acquisition (if it goes wrong, you lose your data)

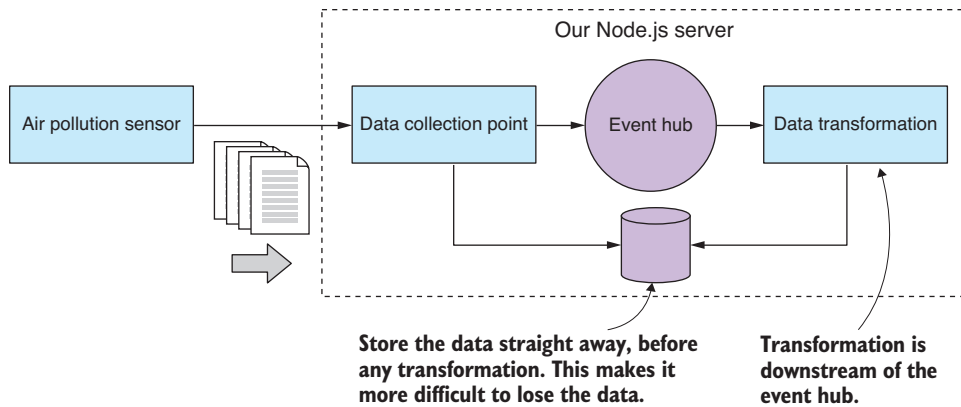


Figure 12.17 Data transformation is downstream from storage (a safer way to manage your data acquisition).

We could put code like this directly in our data collection point before we store the data, as shown in figure 12.16. Obviously, I don't recommend this because it puts us at risk of data loss should anything go wrong with the data transformation (and I've been around long enough to know that something always goes wrong).

To properly mitigate this risk using what I believe is the safest way to structure this code, we can make our downstream data operations always happen on the other side of the event hub. We store the data quickly and safely before triggering any downstream work. As shown in figure 12.17, subsequent operations independently decide how they want to retrieve the data they need, and they have their own responsibility to safely store any data that has been modified.

The data required by the downstream data operation might be passed through the event itself (as we've done with the incoming-data event), or the operation can be made completely independent and must query the database itself to find its own data.

If you now have modified data that needs to be stored, you could overwrite the original data. I wouldn't recommend this approach, however, because if any latent bugs should manifest, you might find that your source data has been overwritten with corrupted data. A better solution is to have the transformed data stored to a different database collection; at least this provides you with a buffer against data-destroying bugs.

12.12 *Live visualization*

We're finally here at the most exciting part of the chapter, the part that you have been waiting for: let's get live data feeding into a dynamically updating chart.

Figure 12.18 shows what our live data chart looks like. When this is running, you can sit back and watch new data points being fed into the chart each second (based on our accelerated notion of time).

To make our live updating visualization, we must do two things:

- 1 Put the initial data into the chart.
- 2 Feed new data points into the chart as they arrive.

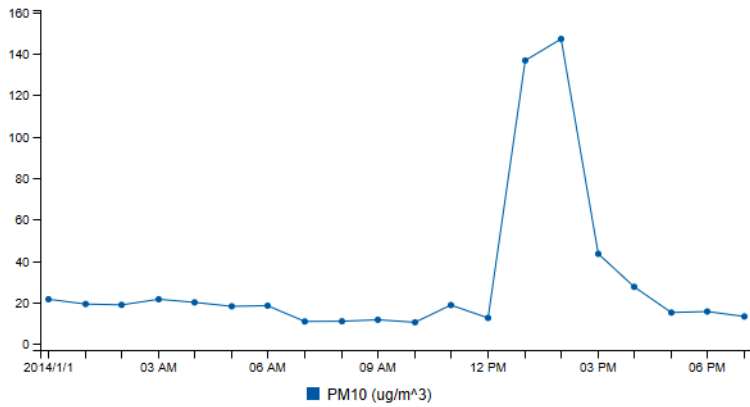


Figure 12.18 The chart we'll be producing from the live data stream

The first one should be familiar to us by now, because we've already seen how to create charts in chapters 10 and 11. Now we'll add the second step into the mix and create a dynamic chart that automatically updates as new data becomes available.

We already have part of the infrastructure we need to make this happen. Let's add a new code module, `update visualization`, to handle the incoming-data event and forward new data points to the browser. See how this fits together in figure 12.19.

I would be remiss if I wrote this chapter and didn't mention `socket.io`. It's an extremely popular library for real-time events, messaging, and data streaming in JavaScript.

`Socket.io` allows us to open a bidirectional communication channel between our server and our web app. We can't use regular sockets to communicate with a sandboxed web app, but `socket.io` uses web sockets, a technology that's built on top of regular HTTP and gives us the data streaming conduit that we need to send a stream of data to the browser. `Socket.io` also has a fallback mode, so if web sockets aren't available, it will gracefully degrade to sending our data using regular HTTP post. This means our code will work on older browsers.

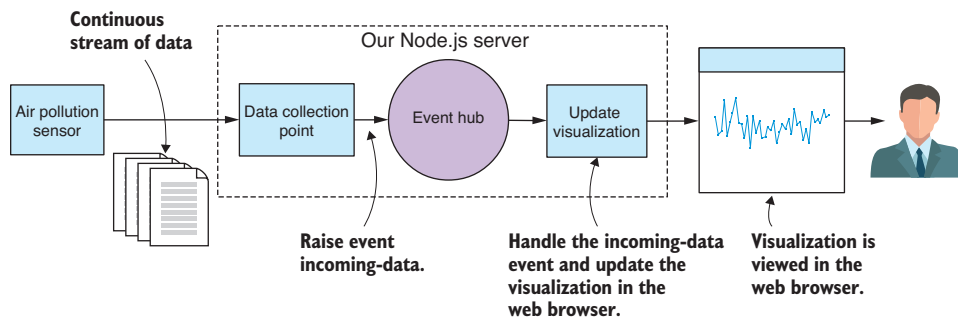


Figure 12.19 Data flowing through to a live visualization

Listing 12.7a shows the code for the web server that hosts our new live visualization. This does three main tasks:

- Serves the assets for the web app itself
- Provides the initial data for the chart
- Registers Socket.io connections with our new code module *update-visualization*

You can see about halfway through the code listing where the web server starts accepting incoming Socket.io connections and registers each with our new *update-visualization* module.

Listing 12.7a Web server for a web app with a live chart for PM10 (listing-12.7 /server/web-server.js)

```
const path = require('path');
const http = require('http');
const socket.io = require('socket.io');
const updateVisualization = require('./update-visualization.js');

function startWebServer (db) {

  const incomingDataCollection = db.collection("incoming");

  const app = express();

  const httpServer = http.Server(app);
  const socket.ioServer = socket.io(httpServer);

  const staticFilesPath = path.join(__dirname, "public");
  const staticFilesMiddleWare = express.static(staticFilesPath);
  app.use("/", staticFilesMiddleWare);

  app.get("rest/data", (req, res) => {
    return incomingDataCollection.find()
      .sort({ _id: -1 })
      .limit(24)
      .toArray()
      .then(data => {
        data = data.reverse(),
        res.json(data);
      })
      .catch(err => {
        console.error("An error occurred.");
        console.error(err);

        res.sendStatus(500);
      });
  });

  socket.ioServer.on("connection", socket => {
    updateVisualization.onConnectionOpened(socket);

    socket.on("disconnect", () => {
      updateVisualization.onConnectionClosed(socket);
    });
  });
}
```

This is a helper function to start a web server that hosts our web app and live data visualization. The database is passed in.

Creates a Socket.io server so that we have a streaming data connection with the web app

Defines a REST API to deliver data to the web app and its visualization

Queries the database for records

Sends the data to the web app

Keeps track of connections and disconnections. We want to be able to forward incoming data to the web app.

```

    });

    httpServer.listen(3000, () => { // Start the server.
      console.log("Web server listening on port 3000!");
    });
  };

  module.exports = startWebServer;

```

Listing 12.7b shows the code for our new update-visualization module, which tracks all open connections, because there could be multiple instances of our web app connected at any one time. Notice where it handles the incoming-data event; here we call `socket.emit` to forward each packet of data to the web app. This is how new data points are sent to the web app to be added to the chart.

Listing 12.7b Forwarding incoming data to the web app (listing-12.7/server/update-visualization.js)

```

const eventHub = require('./event-hub.js');

const openSockets = []; ← This is an array that tracks currently
                           open Socket.io connections.

function onConnectionOpened (openedSocket) { ← This callback function is
  openSockets.push(openedSocket);              invoked when a Socket.io
};                                              connection has been opened.

function onConnectionClosed (closedSocket) { ← This callback function is
  const socketIndex = openSockets.indexOf(closedSocket);
  if (socketIndex >= 0) {
    openSockets.splice(socketIndex, 1);
  }
};                                              invoked when a
                                              Socket.io connection
                                              has been closed.

eventHub.on("incoming-data", (id, incomingData) => {
  for (let i = 0; i < openSockets.length; ++i) { ← For each web app that
    const socket = openSockets[i];                has connected ...
    socket.emit("incoming-data", incomingData);    ...forwards the incoming
  }                                                 data to the web app
});

module.exports = {
  onConnectionOpened: onConnectionOpened,
  onConnectionClosed: onConnectionClosed
}

```

We also need to look at what is happening in the code for the web app. You can see in listing 12.7c that it's mostly the same as what you'd expect to see in a C3 chart (for a refresher, see chapter 10). This time, in addition, we're creating a socket.io instance and receiving incoming-data events from our web server. It's then a simple job to add the incoming-data point to our existing array of data and load the revised data using the C3 load function. C3 conveniently provides an animation for the new data, which gives the chart a nice flowing effect.

Listing 12.7c Adding new data to the chart as it arrives (listing-12.7/server/public/app.js)

```

function renderChart (bindto, chartData) {
  var chart = c3.generate({
    bindto: bindto,
    data: chartData,
    axis: {
      x: {
        type: 'timeseries',
      }
    }
  });
  return chart;
};

$(function () {
  var socket = io();

  $.get("/rest/data")
    .then(function (data) {
      var chartData = {
        xFormat: "%d/%m/%Y %H:%M",
        json: data,
        keys: {
          x: "Date",
          value: [
            "PM10 (ug/m^3)"
          ]
        }
      };

      socket.on("incoming-data", function (incomingDataRecord) {
        chartData.json.push(incomingDataRecord);
        while (chartData.json.length > 24) {
          chartData.json.shift();
        }
        chart.load(chartData);
      });
    })
    .catch(function (err) {
      console.error(err);
    });
});

```

Makes the socket.io connection to the server

Hits the REST API and pulls down the initial data from the server

Sets up chart data that we can update as new data comes down the wire

Handles data that's incoming over the socket.io connection

Does the initial render of the chart

Adds the incoming data to our existing chart data

Removes the oldest data records

Keeps only the most recent 24 hours of records

Reloads the chart's data

One last thing to take note of is how we make Socket.io available to our web app. You can see in listing 12.7d that we're including the socket.io client's JavaScript file into the HTML file for our web app. Where did this file come from?

Well, this file is automatically made available and served over HTTP by the Socket.io library that we included in our server application. It's kind of neat that it's made available like magic, and we don't have to install this file using Bower or otherwise manually install it.

Listing 12.7d Socket.io is automatically available to the client by the server (listing-12.7/server/public/index.html)

```
<!doctype html>
<html lang="en">
  <head>
    <title>Live data visualization</title>

    <link href="bower_components/c3/c3.css" rel="stylesheet">
    <link href="app.css" rel="stylesheet">

    <script src="bower_components/jquery/dist/jquery.js"></script>
    <script src="bower_components/d3/d3.js"></script>
    <script src="bower_components/c3/c3.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="app.js"></script>
  </head>
  <body>
    <div>
      No need to refresh this web page,
      the chart automatically updates as the data
      flows through.
    </div>
    <div id='chart'></div>
  </body>
</html>
```

Includes Socket.io
into the HTML file
for our web app

When you run the code for listing 12.7, keep in mind one caveat: each time you run it fresh (the mock sensor and the server), please reset your incoming MongoDB collection each time (you can remove all documents from a collection using Robomongo). Otherwise, your live chart will come out wonky due to the chronological nature of the data and the fact that we're replaying our fake data. This is an artifact of the way we've set up our development framework with a mock sensor and fake data. This won't be an issue in production. This is a pain during development, so for continued development, you might want to have an automatic way to reset your database to starting conditions.

Well, there you have it. We've built a complete system for processing a continuous feed of live data. Using this system, we can monitor air quality, and hopefully we can be better prepared for emergencies and can respond in real time. You can find the full code under the *complete* subdirectory of the GitHub repo for chapter 12. It brings together all the parts we've discussed in this chapter and combines them into a cohesive functioning system.

The work we've done in this chapter has been a major step toward a full production system, but we're not quite there yet. We still have many issues to address so that we can rely on this system, but we'll come back and discuss those in chapter 14. Let's take a break from the serious stuff, and in chapter 13 we'll upgrade our visualization skills with D3.

Summary

- You learned how to manage a live data pipeline.
- You worked through examples of sending and receiving data through HTTP post and sockets.
- We refactored our code to extract a simple configuration file.
- We brought in an event-based architecture to our app using Node.js' EventEmitter to add a simple event hub for our server.
- We used the cron library to create time-based scheduled jobs.
- We explored using Socket.io for sending data to a live updating C3 chart.

Data Wrangling with JavaScript

Ashley Davis



Why not handle your data analysis in JavaScript? Modern libraries and data handling techniques mean you can collect, clean, process, store, visualize, and present web application data while enjoying the efficiency of a single-language pipeline and data-centric web applications that stay in JavaScript end to end.

Data Wrangling with JavaScript promotes JavaScript to the center of the data analysis stage! With this hands-on guide, you'll create a JavaScript-based data processing pipeline, handle common and exotic data, and master practical troubleshooting strategies. You'll also build interactive visualizations and deploy your apps to production. Each valuable chapter provides a new component for your reusable data wrangling toolkit.

What's Inside

- Establishing a data pipeline
- Acquisition, storage, and retrieval
- Handling unusual data sets
- Cleaning and preparing raw data
- Interactive visualizations with D3

Written for intermediate JavaScript developers. No data analysis experience required.

Ashley Davis is a software developer, entrepreneur, author, and the creator of Data-Forge and Data-Forge Notebook, software for data transformation, analysis, and visualization in JavaScript.

“A thorough and comprehensive step-by-step guide to managing data with JavaScript.”

—Ethan Rivett, Powerley

“Do you still think that you need R and Python skills to do data analysis? This mind-shifting book explains that JavaScript is enough!”

—Ubaldo Pescatore, Datalogic

“Does a fantastic job detailing the wrangling process, the tools involved, and the issues and concerns to expect without ever leaving the JavaScript domain.”

—Alex Basile, Bloomberg

“Excellent real-world examples for full-stack JavaScript developers.”

—Sai Kota, LendingClub

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
mannings.com/books/data-wrangling-with-javascript

ISBN-13: 978-1-61729-484-6
 ISBN-10: 1-61729-484-5



9 781617 294846



\$49.99 / Can \$65.99 [INCLUDING eBook]