CHAPTER 3

How IOT Data Is Stored

Ask any accomplished, professional software engineer, developer, architect, or project lead and they will tell you the key to a well-performing solution starts with a well-designed and tested plan for designing and implementing the data storage element of the project. While there are many other aspects that are equally important to quality and success, the data storage element is definitely among the top contenders for success.

The same is true for IOT solutions. This goes beyond the initial question of where to store the data. To be successful, your plan for developing the solution must also consider what will be stored and, more importantly, how it will be stored.

I've seen many hobbyist solutions that store data in a variety of ways. Some if not most of the solutions work well and have little issue related to the data—that is, until there is a problem such as needing to modify the data, recover the data, or add features to store other data. In these cases, it rapidly becomes clear the data storage component is not up to the task.

Sometimes the result is a need to change how the data is stored, making it incompatible with earlier versions of the software.[1] Other times, it requires a redesign with far more effort than should be needed or even expected, leading to delays completing the project. The root cause of these maladies is typically a poorly or under-designed data storage component.

For example, consider the implications if a solution stored data as text in a file (like a log). Sure, the data is there and you can easily write a program to read it, but the solution doesn't scale to cases where the data gets very large (many thousands or even millions of rows). Furthermore, storing the data in a file means every piece of data must be converted to a string, requiring conversion back to its original type before any mathematical operations are performed. Perhaps worse, there is no easy way to perform any sort of ad hoc query on the data. That is, to perform a query, you must augment the program code used to read the data, writing specialized code to examine the data.

In situations like this, more thought needs to be put into how the data will be used as well as how to store the data. You will examine these topics in more detail in this chapter.

Note The examples are simplistic in an effort to[2] make them easier to follow. A brief overview of the Arduino IDE and program was presented in the previous chapter.

Let's begin with a review of how IOT solutions, specifically distributed IOT solutions, are formed from several types of nodes in network architectures.

[1]Sound familiar? I've had this happen with far too many applications. While it is not always possible, a good data storage design should be extensible.

[2]Sometimes sacrificing efficiency or preferred techniques for easier-to-read code.

Distributed IOT

IOT solutions can be designed and assembled in many ways. Some solutions use a single component design housing all the hardware in a single box. While this design philosophy seems to be common for early IOT solutions, there is another design philosophy that enables more versatility, expandability, and features than a single hardware solution can provide.

This philosophy is borrowed (somewhat) from sensor networks where the solution is composed of multiple, distributed components. In a distributed solution, the components communicate with each other using one or more network protocols. You saw this in the previous chapter where we discussed how data is collected and passed among the nodes in the network.

In this section, we discuss a distributed IOT solution. These solutions are composed of one or more data collectors with one or more sensors using a communication method or protocol to transmit the data. As mentioned, the communication method can use a device such as a microcontroller (for example, an Arduino), an embedded system, or even a small-footprint computer such as a Raspberry Pi.

Typically, the data collectors (called *sensor nodes* in sensor networks) are designed for unattended operation; they're sometimes installed on mobile objects or in locations where wired communication is impractical. In these situations, data collectors can be designed to operate without being tethered to a power (run off a battery or solar power) or communication source (using a wireless mechanism). The receivers of the data collectors can be nodes that process the data and store the data (data aggregator node) or a single database server.

While you saw an overview of each of these types of nodes in Chapter 1, this section presents more details about how each node is used to form a network to gather, transmit, augment, and store data. As you will see, I've divided some of the categories to further define the types of nodes.

Figure 3-1 shows how each type of node would be used in a fictional IOT solution. We will discuss each in more detail in the following sections.

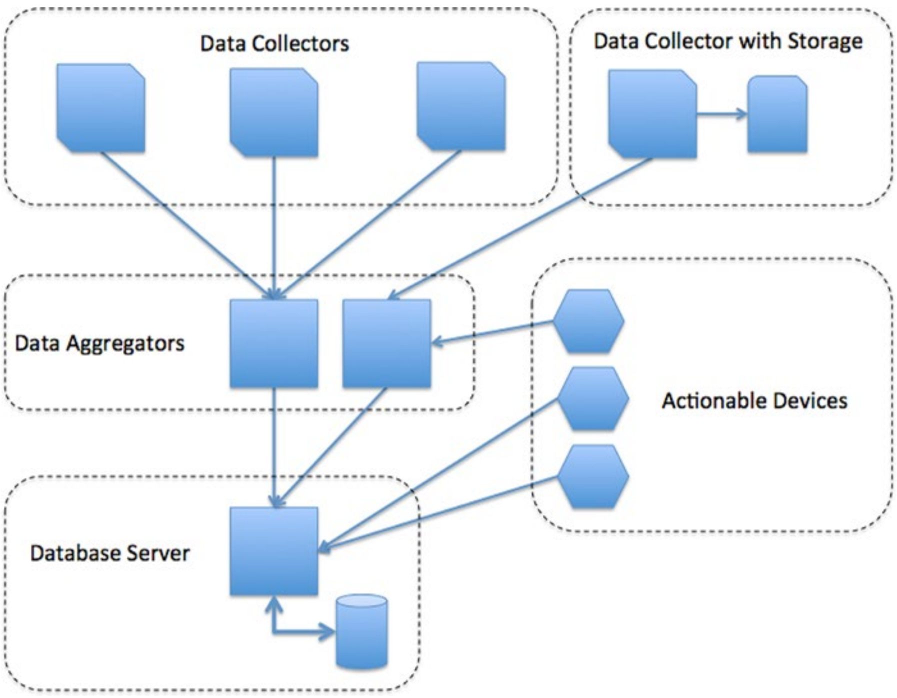

Figure 3-1. *IOT distributed network nodes*

In this example, several data collector nodes at the top send data wirelessly to data aggregator nodes (sometimes called *data nodes*) in the middle. The data node collects the data and saves it to a secure digital card and then communicates with a database server via a wired computer network to store the data. Storing the data on the intermediate data aggregator nodes ensures that you won't lose any data if your database server fails or the network goes down.

Data Collectors

At the lowest (or leaf) level of the network is a data collector. It has at least one sensor and a communication mechanism, typically a wireless protocol. These nodes don't store or manipulate the captured data in any way—they simply pass the data to another node in the network.

Data Collectors with Storage

The next type of node is a data collector node that stores data. While these nodes may send the data to another node, typically they're devices that save the data to a storage mechanism such as a data card, to a database via a link with a desktop or server computer, or directly to a visual output device like an LCD screen, a panel meter, or LED indicators.

Data nodes require a device that can do a bit more than simply pass the data to another node. They need to be able to record or present the data. This is an excellent use for a microcontroller, as you'll see in later chapters. Data nodes can be used to form autonomous or unattended sensor networks that record data for later archiving.

For example, consider a fish or garden pond. There are many commercial pond-monitoring systems that employ self-contained sensor devices with multiple sensors that send data to a data node; the user can visit the data node and read the data for use in analysis on a computer.

Actionable Device

An actionable device is another node that is similar to a data collector and indeed may have data collection features. However, unlike data collectors that only observe and send data, actionable devices can be controlled directly or given commands to execute. For example, a camera with pan and tilt capabilities can produce a video stream or still photos of a wide area as well as receive commands for the pan and tilt features.

Typically, actionable devices require another node in the network to receive and transmit the commands. This could be a forward-facing (as in the Internet) computer or microcontroller or a remotely mounted control panel such as a tablet.

Data Aggregators

Another type of node is an aggregate node. These nodes typically employ a communication device and a recording device (or gateway) and no sensors. They're used to gather data from one or more data collectors or other data aggregator nodes. In the examples discussed thus far, the monitoring system would have one or more aggregator nodes to read the data from sensors.

Data aggregators can be used to augment the data either by adding logic that categorizes the data, adding date and time information, or even performing transformations on the data before it is saved.

Data aggregators can also be used to store a temporary copy of the data before it is sent to a database server. This allows for some minor recoverability or continued data collection should the network encounter failure in one or more nodes or even the communication protocols employed.

Database Server

The database server node is exactly what it sounds like—a computer hosting a database server that can be used to save and retrieve data from the rest of the nodes in the network. While some solutions use a dedicated database server (the networks I design use such), some solutions that support Internet-accessible features such as a web site or control panel will place the database server on the same machine as the application (for example, web server).

However, this is generally considered a potential vulnerability. That is, if the web server is compromised, the likelihood of the database server being compromised is considerable. It is best to keep the database server on a separate node if possible. This vulnerability is lessened if the solution is isolated from other networks or the Internet.

Now that you know what nodes make up a distributed IOT solution, let's look at how you can store data throughout the network. I focus on the database server option last but present a few alternatives first since, depending on the needs of the solution, some local storage may be warranted. For example, storing a copy of the actions taken on the data locally may help diagnose problems.

■ **Note** Since most data collected is sensor data, the following storage options use sensor data to illustrate the concepts.

Local On-Device Storage

Storing data on a local device such as an SD card, hard drive, electronic memory, and so on, can be complicated depending on what the data represents. For example, sensor data can come in several forms. Sensors can produce numeric data consisting of floating-point numbers or sometimes integers. Some sensors produce more complex information that is grouped together and may contain several forms of data. Knowing how to interpret the values read is often the hardest part of using a sensor. In fact, you saw this in a number of the sensor node examples. For example, the temperature sensors produced values that had to be converted to scale to be meaningful.

Although it is possible to store all the data as text, if you want to use the data in another application or consume it for use in a spreadsheet or statistical application, you may need to consider storing it either in binary form or in a text form that can be easily converted. For example, most spreadsheet applications can easily convert a text string like `123.45` to a float, but they may not be able to convert `12E236` to a float. On the other hand, if you plan to write additional code for your Arduino sketches or Raspberry Pi Python scripts to process the data, you may want to store the data in binary form to avoid having to write costly (and potentially slow) conversion routines.

But that is only part of the problem. Where you store the data is a greater concern. You want to store the data in the form you need but also in a location (on a device) that you can retrieve it from and that won't be erased when the host is rebooted. For example, storing data in main memory on an Arduino is not a good idea. Not only does it consume valuable program space, but also it is volatile and will be erased when the Arduino is powered off.

The following sections examine several options for storing data locally. I begin with examples for the Raspberry Pi and Arduino. However, similar platforms offer the same if not similar options.

Local Storage on the Raspberry Pi

The Raspberry Pi offers a number of options for local storage. You can easily create a file and store the data on the root partition or in your home directory on the SD card. This is nonvolatile and does not affect the operation of the Raspberry Pi operating system. The only drawback is that it has the potential to result in too little disk space if the data grows significantly. But the data would have to grow to nearly 2GB (for a 2GB SD card) before it would threaten the stability of the operating system (although that can happen).

It is also possible you could have a removable drive such as a USB thumb drive or even a USB hard drive attached. Once the device and drive partitions are mounted, you can read and write files on them from the Raspberry Pi. You will see this in Chapter 5 when you discover how to build a database server using a Raspberry Pi.

Because the Raspberry Pi is effectively a personal computer, it has the capability to create, read, and write files. Although it may be possible to use an EEPROM connected via the GPIO header, given the ease of programming and the convenience of using files, there is little need for another form of storage.

The Raspberry Pi can be used with a number of programming languages. One of the most popular languages is Python. Working with files in Python is easy and is native to the default libraries. This means there is nothing you need to add to use files.

The following example demonstrates the ease of working with files in Python. One thing you will notice is that it doesn't matters where the file is located—on the SD card or an attached USB drive. You only need know the path to the location (folder) where you want to store data and pass that to the `open()` method.

■ **Tip** The online Python documentation explains reading and writing files in detail (`http://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files`).

Writing Data to Files

This example demonstrates how easy it is to use files on the Raspberry Pi with Python. I will demonstrate how to read and write files from the logged-in user's home directory. This does not require any additional hardware or software libraries. Thus, you can execute this example on any Raspberry Pi (or any Linux-compatible system). In this example, the file you will access acts like a log. That is, you always write new data to the end of the file.

You begin by creating a file to contain the Python commands. If you do not know Python, don't worry because the commands are easy to understand and for the most part are intuitive in their usage. If you have ever written a program to read files (or a script or a command/batch file), this code will look familiar.

If you haven't used a Raspberry Pi but have a computer running Mac, Linux, or Windows, you can execute this example there as well. Just remember to change the path to the file you want to read and write to something appropriate for your system.

Start by powering on and logging into your Raspberry Pi. Then open a new file with the following command (or similar) in your home directory or some place you have read and write privileges. You can use whatever editor you want. Listing 3-1 shows the code for the example.

```
nano log_file_example.py
```

■ **Tip** Name the file with a `.py` extension to indicate that it is a Python script. Enter the code in Listing 3-1 in the file.

Listing 3-1. Log File Example (Raspberry Pi)

```
from __future__ import print_function
import datetime   # date and time library

# We begin by creating the file and writing some data.
log_file = open("log.txt", "a+")
for i in range(0,10):
  log_file.write("%d,%s\n" % (i, datetime.datetime.now()))
log_file.close()

# Now, we open the file and read the contents printing out
# those rows that have values in the first column > 5
log_file = open("log.txt", "r")
rows = log_file.readlines();
for row in rows:
  columns = row.split(",")
  if (int(columns[0]) > 5):
    print(">", row, end="")

log_file.close()
```

In this example, you first import the `datetime`. You use the `datetime` to capture the current date and time. Next, you open the file (notice that you are using the current directory since there is no path specified), write ten rows to the file, and then close the file.

Notice the `open()` method. It takes two parameters—the file path and name and a mode to open the file. You use `"a+"` to append to the file (`a`) and create the file if it does not exist (`+`). Other values include `r` for reading and `w` for writing. Some of these can be combined. For example, `"rw+"` creates the file if it does not exist and allows for both reading and writing data.

■ **Note** Using write mode truncates the file. For most cases in which you want to store sensor samples, you use append mode.

Next, you close the file and then reopen it for reading. This demonstrates how you can read a file and search it for data. In this case, you read all the rows and then for each row divide (using the `split()` function) it into columns. Notice in the line that writes the data the columns are separated with a comma.

In this case, you are looking for any row that has the first column greater than five. Since the file is a text file and therefore every row read is a string, you use the `int()` function to convert the first column to an integer. Once you find the row, you print it out.

It is at this point I should mention that reading the file requires an intimate knowledge of its layout (composition) with respect to the number of columns, data types, and so on. Without this knowledge, you cannot write a program to read the data reliably.

If you are following along, go ahead and run the script. To execute the file, use the following command:

```
python ./log_file_example.py
```

If you get errors, check the code and correct any syntax errors. If you encounter problems opening the file (you see I/O errors when you run the script), try checking the permissions for the folder you are using. Try running the script a number of times and then display the contents of the file. Listing 3-2 shows the complete sequence of commands for this example running three times in succession.

Listing 3-2. Log File Example Output (Raspberry Pi)

```
$ python ./log_file_example.py
> 6,2015-10-14 20:42:33.063794
> 7,2015-10-14 20:42:33.063799
> 8,2015-10-14 20:42:33.063804
> 9,2015-10-14 20:42:33.063808
$ python ./log_file_example.py
> 6,2015-10-14 20:42:33.063794
> 7,2015-10-14 20:42:33.063799
> 8,2015-10-14 20:42:33.063804
> 9,2015-10-14 20:42:33.063808
> 6,2015-10-14 20:42:38.128724
> 7,2015-10-14 20:42:38.128729
> 8,2015-10-14 20:42:38.128734
> 9,2015-10-14 20:42:38.128739
$ python ./log_file_example.py
> 6,2015-10-14 20:42:33.063794
> 7,2015-10-14 20:42:33.063799
> 8,2015-10-14 20:42:33.063804
> 9,2015-10-14 20:42:33.063808
> 6,2015-10-14 20:42:38.128724
> 7,2015-10-14 20:42:38.128729
> 8,2015-10-14 20:42:38.128734
> 9,2015-10-14 20:42:38.128739
> 6,2015-10-14 20:42:39.262215
> 7,2015-10-14 20:42:39.262220
> 8,2015-10-14 20:42:39.262225
> 9,2015-10-14 20:42:39.262230
```

Did you get similar results? If not, correct any errors and try again until you do. Notice how the first time I ran the script I got three rows, the next six, and then nine rows. This is because the first part of the script appends data to the end of the file. If you want to start over, simply delete the file created.

As you can see from this simple example, it is easy to read and write log files using Python.

Local Storage on the Arduino

Although it is true that the Arduino has no onboard storage devices,[3] there are two ways you can store data locally on the Arduino. You can store data in a special form of nonvolatile memory or on an SD card hosted via either a special SD card shield or an Ethernet shield (most Ethernet shields have a built-in SD card drive).

■ **Note** If you are truly inventive (or perhaps unable to resist a challenge), you can use some of the communication protocols to send data to other devices. For example, you could use the serial interface to write data to a serial device.

[3]Except for the new Arduino Yún, which has an SD drive and USB ports for connecting external devices. The Yún and newer boards are sure to be a game changer for the Arduino world.

The following sections discuss each option in greater detail. Later sections present small projects you can use to learn how to use these devices for storing data.

Nonvolatile Memory

The most common form of nonvolatile memory available to the Arduino is electrically erasable programmable read-only memory (EEPROM—pronounced "e-e-prom" or "double-e prom"). EEPROMs are packaged as chips (integrated circuits). As the name suggests, data can be written to the chip and is readable even after a power cycle but can be erased or overwritten.

Most Arduino boards have a small EEPROM where the sketch is stored and read during power up. If you have ever wondered how the Arduino does that, now you know. You can write to the unused portion of this memory if you desire, but the amount of memory available is small (512KB). You can also use an EEPROM and wire it directly to the Arduino via the I2C protocol to overcome this limitation.

Writing to and reading from an EEPROM is supported via a special library that is included in the Arduino IDE. Because of the limited amount of memory available, storing data in the EEPROM memory is not ideal for most data nodes. You are likely to exceed the memory available if the data you are storing is large or there are many data items per sample.

You also have the issue of getting the data from the EEPROM for use in other applications. In this case, you would have to build not only a way to write the data but also a way to read the data and export it to some other medium (local or remote).

That is not to say that you should never use EEPROM to store data. Several possible reasons justify storing data in EEPROM. For example, you could use the EEPROM to temporarily store data while the node is offline. In fact, you could build your sketch to detect when the node goes offline and switch to the EEPROM at that time. This way, your Arduino-based data node can continue to record sensor data. Once the node is back online, you can write your sketch to dump the contents of the EEPROM to another node (remote storage).

A detailed example of how to work with the EEPROM on an Arduino is beyond the scope of this book. But I wanted to discuss this option since it is a viable alternative for the Arduino and for a small amount of data can be handy. However, I have described the process in detail in Chapter 5 of my book *Beginning Sensor Networks Using the Arduino and Raspberry Pi* (Apress, 2014).

SD Card

You can also store (and retrieve) data on an SD card. The Arduino IDE has a library for interacting with an SD drive. In this case, you would use the library to access the SD drive via an SD shield or an Ethernet shield.

Storing data on an SD card is done via files and is similar in concept to the previous example. You open a file and write the data to it in whatever format is best for the next phase in your data analysis. Examples in the Arduino IDE and elsewhere demonstrate how to create a web server interface for your Arduino that displays the list of files available on the SD card.

You may choose to store data to an SD card in situations where your data node is designed as a remote unit with no connectivity to other nodes, or you can use it as a backup logging device in case your data node is disconnected or your data aggregator node goes down. Because the card is removable and readable in other devices, you can read it on another device when you want to use the data.

Using an SD card means you can move the data from the sensor node to a computer simply by unplugging the card from the Arduino and plugging it in to the SD card reader in your computer. Let's see how you can read and write data to an SD card on an Arduino. Listing 3-3 shows a complete example of the log file concept. You will need an SD card formatted as a FAT partition.

STORING DATE AND TIME WITH SAMPLES

The Arduino does not have a real-time clock (RTC) on board. If you want to store your data locally, you have to either store the data with an approximate date and time stamp or use an RTC module to read an accurate date/time value. Fortunately, there are RTC modules for use with an Arduino.

Listing 3-3. Log File Example (Arduino)

```
/**
  Example Arduino SD card log file.

  This project demonstrates how to save data to a
  microSD card as a log file and read it.
*/
#include <SPI.h>
#include <SD.h>
#include <String.h>

// Pin assignment for Arduino Ethernet shield
#define SD_PIN 4
// Pin assignment for Sparkfun microSD shield
//#define SD_PIN 8
// Pin assignment for Adafruit Data Logging shield
//#define SD_PIN 10

File log_file;

void setup() {
  char c = ' ';
  char number[4];
  int i = 0;
  int value = 0;
  String text_string;

  Serial.begin(115200);
  while (!Serial); // wait for serial to load

  Serial.print("Initializing SD card...");

  if (!SD.begin(SD_PIN)) {
    Serial.println("ERROR!");
    return;
  }
  Serial.println("done.");
  // Begin writing rows to the file

  log_file = SD.open("log.txt", FILE_WRITE);
  if (log_file) {
    for (int i=0; i < 10; i++) {
      text_string = String(i);
      text_string += ", Example row: ";
      text_string += String(i+1);
      log_file.println(text_string);
    }
    log_file.close();
  } else {
    Serial.println("Cannot open file for writing.");
  }

  // Begin reading rows from the file

  log_file = SD.open("log.txt");
  if (log_file) {
    // Read one row at a time.
    while (log_file.available()) {
      text_string = String("");

      // Read first column
      i = 0;
      while ((c != ',') && (i < 4)) {
        c = log_file.read();
        text_string += c;
        if (c != ',') {
          number[i] = c;
        }
        i++;
      }
      number[i] = '\0';
      value = atoi(number);

      // Read second column
      c = ' ';
      while (c != '\n') {
        c = log_file.read();
        text_string += c;
      }
      // If value > 5, print the row
      if (value > 5) {
        Serial.print("> ");
        Serial.print(text_string);
      }
    }
    // close the file:
    log_file.close();
  } else {
    // if the file didn't open, print an error:
    Serial.println("Cannot open file for reading");
  }
}

void loop() {
  // do nothing
}
```

■ **Note** I omitted the writing of the date and time. There is a way to do this, but it is quite a bit more code, and I wanted to concentrate on the file operations. To see how to use the `DateTime` library on the Arduino, see `http://playground.arduino.cc/Code/DateTime`.

Notice there is a lot more code here than the Raspberry Pi example. This is because the Arduino code libraries do not have the same high-level primitives as Python. Thus, we have to do a lot of lower-level operations ourselves.

To keep it simple, I put the code in the `setup()` method so that it runs only once. Recall code in the `loop()` method runs repeatedly until the Arduino is powered off (or you initiate low-level code to halt execution or reboot). If you were using this example in an IOT solution, the code for initiating the SD card would remain in the `setup()` method and perhaps even the code to open the file, but the code to write the file would be moved to the `loop()` method to record data read from sensors or other data collectors.

The code example begins with some variables used for selecting the pin used for communicating with the SD card. I've included several popular options. Check the documentation for your hardware to ensure the correct pin is specified.

Next, you will see code for opening the file and writing several rows. In this case, I simply write a simple text string using the value of the counter followed by a bit of short text. As I mentioned, I did not record date and time information in this example because the Arduino does not include an RTC (although newer boards may include an RTC). Notice I open the file, write the rows, and then close the file.

Next is a block of code to read rows from the file. Like the Raspberry Pi example, we have to read the row and split the columns ourselves. However, in this case, we must read a character at a time. Thus, I use a loop to do so and terminate when the comma is located. The second part of the loop reads the remaining text from the row until the newline character is found.

I then check the value of the first column (after converting it to an integer), and if > 5, I print out the row to the serial monitor. Let's see this code in action. Listing 3-4 shows the output of the code as seen in the serial monitor.

Listing 3-4. Log File Example Output (Arduino)

```
Initializing SD card...done.
> 6, Example row: 7
> 7, Example row: 8
> 8, Example row: 9
> 9, Example row: 10
Initializing SD card...done.
> 6, Example row: 7
> 7, Example row: 8
> 8, Example row: 9
> 9, Example row: 10
> 6, Example row: 7
> 7, Example row: 8
> 8, Example row: 9
> 9, Example row: 10
Initializing SD card...done.
> 6, Example row: 7
> 7, Example row: 8
> 8, Example row: 9
> 9, Example row: 10
> 6, Example row: 7
> 7, Example row: 8
> 8, Example row: 9
> 9, Example row: 10
> 6, Example row: 7
> 7, Example row: 8
> 8, Example row: 9
> 9, Example row: 10
```

As you can see, the output is similar to the Raspberry Pi example. Clearly, either can be used to store data locally in files, but the Arduino takes a bit more work. Fortunately, the Arduino IDE includes a well-designed SD card library.

In fact, there are a number of other functions in the SD card library for dealing with files. For example, you can list the files on the card, create folders, and even truncate or delete the contents of a file. You may find this code useful in working with the previous code. The following shows how easy it is to truncate a file. If you want to use this, place it in the code before the file is opened the first time.

```
if (SD.remove("log.txt")) {
  Serial.println("file removed");
}
```

■ **Tip** For more information about using SD cards on the Arduino, see the Arduino online reference guide (http://arduino.cc/en/Reference/SDCardNotes).

Now that you've seen some of the options for storing data locally on the data collector nodes, the following section discusses sending data to data aggregators for either local or remote storage. Remote storage in this case is typically a database server.

Passing the Buck to Aggregators

Recall that a data aggregator is a special node designed to receive information from multiple sources (data collectors or sensors) and store the results either locally or remotely. The source data can originate from multiple sensors on the node itself, but more often the data aggregator receives information from multiple data collector nodes that are not attached directly to the aggregate node (they may be connected via a low-power, low-overhead communication protocol such as provided by XBee modules).

WHAT'S AN XBEE?

XBee is a brand name of Digi International for small, self-contained, modular, cost-effective components that use radio frequency (RF) to exchange data from one XBee module to another. XBee modules transmit on 2.4GHz or long-range 900MHz and have their own network protocols (ZigBee).

Although the XBee isn't a microcontroller, it does have a limited amount of processing power that you can use to control the module. One of these features, the sleep mode, can help extend battery life for battery-powered (or solar-powered) sensor nodes. You can also instruct the XBee module to monitor its data pins and transmit the data read to another XBee module. Thus, you can use an XBee module to send data from one or more sensors to a data aggregator node.

In some IOT solutions, sensors are hosted by other nodes and placed in remote locations. The data aggregator node is connected to the data collector nodes via a wired or wireless connection. For example, you may have a sensor hosted on a low-power Arduino in one location and another sensor hosted on a Raspberry Pi in another location, both connected to your data aggregator using XBee modules. Except for the limitations of the network medium chosen, you can have dozens of nodes feeding sensor data to one or more data aggregator nodes.

The use of data aggregator nodes has several advantages. If you are using a wireless technology such as ZigBee with XBee modules, data aggregator nodes can permit you to extend the range of the network by placing the data aggregator nodes nearest the sensors. The data aggregator nodes can then transmit the data to another node such as a database server via a more reliable medium.

For example, you may want to place a data aggregator node in an outbuilding that has power and an Ethernet connection to collect data from remote data collector nodes located in various other buildings.

■ **Note** In this case, I mean the closest point to the sensor nodes that is still within range of the wireless transmission media (such as ZigBee used on XBee modules).

Data aggregator nodes can also permit you to move the logic to process a set of sensors to a more powerful node. For example, if you use sensors that require code to process the values, you can use a data aggregator node to receive the raw data from those sensors, store it, and calculate the values at a later time.

Not only does this ensure that you have code in only one location, but it also allows you to use less sophisticated (less powerful) hosts for the remote sensors. That is, you could use less expensive or older Arduino boards for the sensors and a more powerful Arduino for the data aggregator node. This has the added advantage that if a remote sensor is destroyed, it is not costly to replace.

Recall also that you have to decide where you want to store your sensor data. Data aggregator nodes either can store the data locally on removable media or an onboard storage device (local storage) or can transmit the data to another node for storage (remote storage). The choice of which to use is often based on how the data will be consumed or viewed.

For example, if you want to store only the last values read from the sensors, you may want to consider some form of visual display or remote-access mechanism. In this case, it may be more cost effective and less complicated to use local storage, storing only the latest values.

On the other hand, if you require data values recorded over time for later processing, you should consider storing the data on another node so that the data can be accessed without affecting the sensor network. That is, you can store the data on a more robust system (say, a personal computer, server, or cloud-based service) and further reduce the risk of losing data should the aggregate node fail.

The nature of the local storage is a limiting factor in what you can do with a local-storage data aggregator node. That is, if you want to process the data at a later time, you would choose a medium that permits you to retrieve the data and move it to another computer.

This does not mean the local-storage data aggregator is a useless concept. Let's consider the case where you want to monitor temperature in several outbuildings. You are not using the data for any analysis but merely want to be able to read the values when it is convenient (or required).

One possible solution is to design the local-storage data aggregator node with a visual display. For example, you can use an LCD to display the sensor data. Of course, this means the data aggregator node must be in a location where you can get to it easily.

But let's consider the case where your data aggregator node is also in a remote location. Perhaps it too is in another outbuilding, but you spend the majority of your time in a different location. In this case, a remote-access solution would be best.

The design of such a data aggregator node would require storing the latest values locally, say in memory or EEPROM and, when a client connects, displaying the data. This is a simple and elegant solution for a local-storage data aggregator node.

However, it is more robust to pass the data from the data collectors to a database server. As you will see in Chapter 5, you can use a Raspberry Pi or similar low-cost board to build a database server and deploy it in your IOT solution.

The data aggregator nodes would then require a library called a *connector* to allow you to write programs (scripts, sketches) to connect to the database server and send the data. In this book, I feature the MySQL database server. In the case of the Raspberry Pi, the database connector I will demonstrate is called Connector/Python and is available from Oracle (`http://dev.mysql.com/downloads/connector/python/`). I will also discuss a database connector for the Arduino, called Connector/Arduino, which I created[4] and is available on GitHub (`https://github.com/ChuckBell/MySQL_Connector_Arduino`) or via the Library Manager in the Arduino IDE. The connector therefore forms a communication pathway to the MySQL server.

Even with the use of the connector, there are many things that you can do with a database server in your solution. Not only does a database server provide a robust storage mechanism, it can also be used to offload some of the data-processing steps from data aggregators. I discuss some of the ramifications and things to consider when using a database server node in your IOT solution.

Database Storage

A database option represents a more stable, more easily scaled, and more easily queried storage option. Indeed, that is what this book is all about—discovering how best to employ a database server in your IOT solution!

In this section, we explore some of the benefits, techniques, and considerations for using a database server in your IOT solution. As you will see, there is a lot of power available to you when you use a database server. I will give a high-level overview of the topics here with a more in-depth, hands-on explanation in Chapter 5. Let's begin by discussing why you would use a database server in your IOT solution.

■ **Tip** While an in-depth, full discussion of database design is beyond the scope of this book, the following text views the subject from a slightly different angle: how you can best design your databases for easy storage and retrieval. Thus, I assume no prior knowledge of database design. If you have database design experience, you may want to skim this section.

---

[4]Officially owned by Oracle but supported by me exclusively.

Database servers provide a structured manner in which to store and retrieve data. Interaction with a database server requires the use of a special set of commands or, more precisely, a special language that expresses storage and retrieval. The language is called Structured Query Language (SQL).[5] Most database servers use a form of SQL for most of their commands. While there are some differences from one database server to another, the syntax and indeed the core SQL commands are similar.

In this book, we use the MySQL database system as the database server. MySQL[6] is the most popular choice for developers because it offers large database system features in a lightweight form that can run on just about any consumer computer hardware. MySQL is also easy to use, and its popularity has given rise to many online and printed resources for learning and using the system.

Benefits

As mentioned, using a database server in your IOT solution has many advantages. Not only does a database server permit structured, robust storage of your data, but it also provides a powerful mechanism for retrieving data.

Consider a solution where you store data in files. As you saw in the previous sections, reading a file and looking for information requires parsing (separating the data elements) the data and then comparing the data to fit the criteria needed. The problem is each time you want to do a search you need to modify your program or script.

This doesn't sound too bad, but consider the possibility that you may need to execute searches (queries) at any time. Furthermore, consider it possible you want to be able to do this without rewriting your code or perhaps you want to allow your users to execute the queries. Clearly, using files or similar storage mechanisms does not easily permit this behavior.

This is one of the most beneficial aspects of using a database server. You can execute a query at any time (ad hoc) and you do not need a special program or need to rewrite anything to use it. I should note that some solutions hard-code their queries in their code, and some would argue that it is the same thing as accessing files. But it isn't.

In the case of a file-based solution, even if you use programming primitives, you still must write code to execute the query, whereas in a database solution, you need only to replace the query statement itself. For example, consider the Raspberry Pi file example in Listing 3-1. The code needed to execute the query (choose only those rows where the value of the first column is greater than five) is several lines long[7] and requires not only choosing the rows that match but also having to read the columns one character at a time.

Now consider the following SQL statement. Don't worry about the details of the command. Consider only that we can express the query as a single statement as follows:

```
SELECT * FROM db1.table1 WHERE col1 > 5;
```

[5] https://en.wikipedia.org/wiki/SQL

[6] http://dev.mysql.com/

[7] The Arduino example is three times the number of lines of code!

Notice the criteria have been moved to the database server. That is, the code doesn't need to read, interpret, and test the values for the column. Rather, that logic is executed on the database server. In fact, the database server is optimized to execute such a query in the most efficient way possible—something that would require considerable work on a file-based solution (but not unheard of). In case you are curious, the code to execute and retrieve the rows is as follows:

```
cur.execute("SELECT * FROM db1.table1 WHERE col1 > 5")
rows = cur.fetchall()
for row in rows:
    print ">", row
```

While this is a trivial example, the point is still valid. Specifically, the database server is a powerful searching tool. Not only does this allow you to simplify your code, it allows much greater expressiveness when constructing queries.

For example, you can construct queries that perform complex mathematical comparisons, text comparisons (even wildcard matching), and date comparison. For instance, you can use date operations to select rows older than N days from time of execution or rows recorded during a specific year, month, day, or hour. Clearly, this is far more powerful than code you write yourself!

Another benefit of using a database server is you can group your data in a logical manner. A database server permits you to create any number of databases for storing data. Typically, you want to create a separate database for each of your IOT solutions. This makes working with the data at a logical level easier so that data for one solution isn't intermixed with data from another. Thus, a single database server can support many IOT solutions.

Techniques

Using a database server in your solution requires using some different techniques than other storage solutions. You have already seen that file-based systems are susceptible to file layout changes (if the file layout changes, so too must the code); the same is not true for databases.

For example, if you need to add a column to the table, you don't have to rewrite code to read the data. Indeed, many changes can be made to the database without affecting the code. While some changes may affect the SQL statements, you have a great deal of freedom in how the data is stored than file-based solutions.

Thus, the biggest change in development is how you work with the database server versus the code itself. That is, code development can proceed somewhat independently of the database development. You can develop the database, its components, and the SQL statements separately from the code. Indeed, you can create a working and tested database component before you develop any of the other nodes!

This may seem like it is more work than the code for file-based solutions, but it really isn't for two reasons. First, you can test your SQL statements in isolation with test data. This means you do not need to have your entire IOT solution up and running, which makes it easier to develop. Second, and related, is you can execute your SQL statements repeatedly to ensure you get the correct data, which makes it easier to separate the data from the code and thereby makes the solution easier to maintain. That is, if something changes, you can simply change the SQL statements rather than rewrite the code.

But this does not mean you do not need to spend time designing and testing the database design. On the contrary, to reap these benefits, your database should be designed well. Database design can be complex if the data itself or your use of the data is complex. Fortunately, for most IOT solutions, this isn't a problem.

Finally, working with a database server allows you to quickly set up test data, manipulate it, and refresh it. Again, this is possible with other storage solutions, but it is so much easier with database servers. Thus, the technique for setting up test data permits you to ensure your queries return precisely what you expect. This is because you know the input (the sample data) and can easily and manually determine what the results should be.

After all, using a database server over a file-based or memory-based storage solution changes how you develop your solution by making it easier to work with the data through testing the SQL statements outside of the applications or deploying and executing the network nodes.

Considerations

Since you are reading this book, you are most likely convinced or nearly convinced you want to employ a database server in your IOT solution. But perhaps you're wondering what the ramifications or limitations are using a database server in your solution.

Perhaps the most important consideration is you need a platform on which to host the database server. If your solution employs a computer, you have everything you need. MySQL runs on commodity hardware, and for small solutions like an IOT solution, even the most basic computer is more than adequate.

However, what if you do not have a computer? In this case, you need to add hardware to host the database server. Fortunately, MySQL runs on most low-cost computer boards such as the Raspberry Pi, Beaglebone Black, pcDuino, and even the newest Intel IOT boards. While this means another node in your network, you have seen that this node fits very well into the plan (a database node). I give a complete tutorial on building a MySQL database node in Chapter 5.

Aside from adding a new node, other considerations include using SQL in your code. For this, we use a connector to connect to the MySQL server via an Ethernet connection, send queries to the server for execution, and then retrieve and process the results. If you do not know SQL, you will have to learn how to form queries. However, as you will see in Chapters 5 and 6, SQL statements are not difficult to learn and use.

Another consideration is storing data in a database server requires a good design confined to the database server terminology and features. More specifically, you have to form the layout of the tables to include selecting the correct data type from a long list of types available.

Part of this process (called *database design*) concerns designing the queries themselves or, more specifically, to design the queries so that they return exactly what you want. Thus, queries should be tested on the database server (or through a client connection, as you will see in Chapter 5) to ensure your SQL statements are correct and that there are no surprises should unusual data appear. As you saw in the previous section, the advantage is easier development of the layer of the solution that deals with the data.

When you design your tables, you should keep a few things in mind. First, consider what data types are needed for storing your samples. You should consider not only how many values each sample contains but also their format (data type). The basic data types available include integer, float, double, character, and Boolean. There are many others, including several for dates and times, as well as binary large objects (blobs) for storing large blocks of data (like images), large texts (the same as blobs, but not interpreted as binary), and much more.

You can also consider adding columns such as a timestamp field, the address of the data collector node, perhaps a reference voltage, and so on. Write all of these down, and consider the data type for each.

■ **Tip** See the online MySQL Reference Manual for a complete list and discussion of all data types (`http://dev.mysql.com/doc/refman/5.7/en/`).

What may be a consideration for the database design is the physical storage required for the database. If your IOT solution uses nodes that do not have any physical storage, you should add one. Not only does physical storage mean the data is not susceptible to node failure (say if memory gets erased), but it also permits you to use media with large storage capacities. While you can use a secure digital memory card, it is best to use a solid-state or older spindle disk.

Maintenance of the data is another consideration, but perhaps less so than the others mentioned. More specifically, even file-based solutions require maintenance or at least facilities for conducting maintenance such as backup and recovery. For file-based solutions, this can be simply a matter of copying files. For a database solution, backup and recovery are a bit more complicated.

Fortunately, you can use utilities such as mysqlpump[8] (or the older mysqldump[9]) or MySQL Utilities[10] to make logical backups of the data. That is, these utilities produce a file of SQL commands you can replay to create and store the data. For physical backups (at the byte level), you would have to use a commercial application such as MySQL Enterprise Backup to back up and restore the data.

Finally, security can be a consideration if your database server node is visible from outside your network. In this case, you must take care to ensure not only is the platform access (for example, the Raspberry Pi) secured but also all database security is properly designed so that users who have access to the database server have only enough permission to execute the queries for the solution and nothing more. In other words, you need to design your solution with security in mind.

Now that you've explored the database node, let's look at some best practices for designing and implementing a distributed IOT network.

Distributed IOT Network Best Practices

If your IOT solution needs to be deployed over a disperse area where the data collectors are physically separated or you want to use commodity, low-cost hardware to develop your solution, you may need to design your IOT solution using a distributed network of nodes.

This section examines some important considerations for planning the network. I discuss placement of the nodes in the network as well as design considerations for data storage. Most of these best practices are data-centric for good reason—the solution is useless without data that is accessible.

Node Placement

When planning your solution, you should consider what data you want to collect. Moreover, you should consider where and how you would collect the data. This includes where the sensors need to be located as well as what data is produced.

Placing the nodes with sensors (data collectors) is likely to be a simple choice—they need to be near the things they are observing. If the data collectors are outside, they may need weatherproof enclosures. Even inside you may need to secure the hardware to avoid accidental tampering (such as small fingers or nosy, curious friends[11]). In fact, I recommend using an appropriate enclosure for each node.

However, where you place the data aggregators may be more problematic. This is normally dictated by the communication mechanism chosen. Recall if you use a low-overhead mechanism such as ZigBee or Bluetooth, you may be limited to a certain range. Thus, the data aggregator needs to be close enough to communicate with the data collectors.

[8]mysqlpump is available in server versions 5.7.8 and newer (`http://dev.mysql.com/doc/refman/5.7/en/mysqlpump.html`).
[9]mysqldump is available in server versions 5.6 and prior (`http://dev.mysql.com/doc/refman/5.7/en/mysqldump.html`).
[10]Using mysqldbexport and mysqldbimport (`http://dev.mysql.com/doc/mysql-utilities/1.6/en/mysqldbexport.html`).
[11]I've had more experiments wrecked by a casual, "Hey, what's this?" inquiry often involving mishandling or relocation of the device.

Furthermore, the data aggregator nodes are best placed where they can communicate to the database server or visualization application (perhaps another node hosting a web server or cloud gateway) via a WiFi or Ethernet network. Thus, depending on how many data collectors you have and their physical proximity to a location where a data aggregator could be placed, you may need to employ multiple data aggregators. If the data collectors are all placed within the range of your low-cost communication mechanism, you may be able to use a single data aggregator.

Perhaps on a lesser scale are the capabilities of the data aggregator hardware. If you choose a platform that can support a limited number of connections, you are therefore limited to how many data collectors the data aggregator can support. Once again, you may need more than one data aggregator to support all the data collectors.

Another node type to consider for node placement is an actionable device. If the device produces data, you may need to connect it to a data aggregator. However, if the device has the ability to be programmed or to run scripts, you may be able to program it to write data directly to the database server.

Finally, placing the database node is less critical since it will be using either a WiFi or Ethernet network. As such, it merely needs to be on the same network (or made accessible from the network that the data aggregators use).

Data Storage

When considering your data storage, you should consider what the data looks like, that is, what data the sensors produce to include the data type and how the data is used.

For example, if the sensor data is a number in a range of values, say -5.0 to +5.0, the numeric value may not be very meaningful. For example, what does it mean when values are closer to -5.0 versus +5.0? You should always store the original value, but you may want to store a representative value. In this case, there may be several thresholds that determine a qualitative value. Consider the following thresholds for the fictional sensor range. In this case, the value is a voltage reading.

- `-5.0` : Error, no signal
- `-4.9` to `-2.0` : Low
- `-2.0` to `0.0` : OK, decreasing
- `0.0` to `+2.0` : OK, increasing
- `+2.0` to `+4.9` : High
- `+5.0` : Error, no signal

Notice there are two error conditions. I've seen this before in other sensors, and it depends on how the sensor is powered or signaled. While it is not unusual to see a value like this, you are not likely to encounter a sensor that has multiple error readings (but I have seen some).

Notice also that there are four distinct thresholds that tell us what the values mean. For example, if the value read is +3.3, we know the data can be interpreted as "high." Thus, we can store the original value in one column and a category in another. For example, we could have one floating-point field (column) and another text column with values of (low, decreasing, increasing, high, and error). We would use a data aggregator or the database server to assign these values. I show you an example of this in Chapter 5.

This information is vital to planning your data aggregator code. You need to understand what the data means and how best to interpret it. In fact, I recommend documenting it in your code as well as your notebook. The information will be vital should you replace the sensor or discover you need to adjust the thresholds.

For example, if the threshold for increasing or decreasing needs to be adjusted to a higher or lower value, values read that are close to the original threshold may need to be modified. If you had not stored the original value, you would have no way to adjust the data you have already stored.

In addition to how to interpret the data, you should consider how the data flows through your network. I like to make a drawing that shows where each type of data originates and how it moves through the network. Figure 3-2 shows an example of a data flow chart. You can use any form you want—from a simple list written in a log book to a graphical picture written in a structured design language like the Unified Modeling Language (UML).[12]

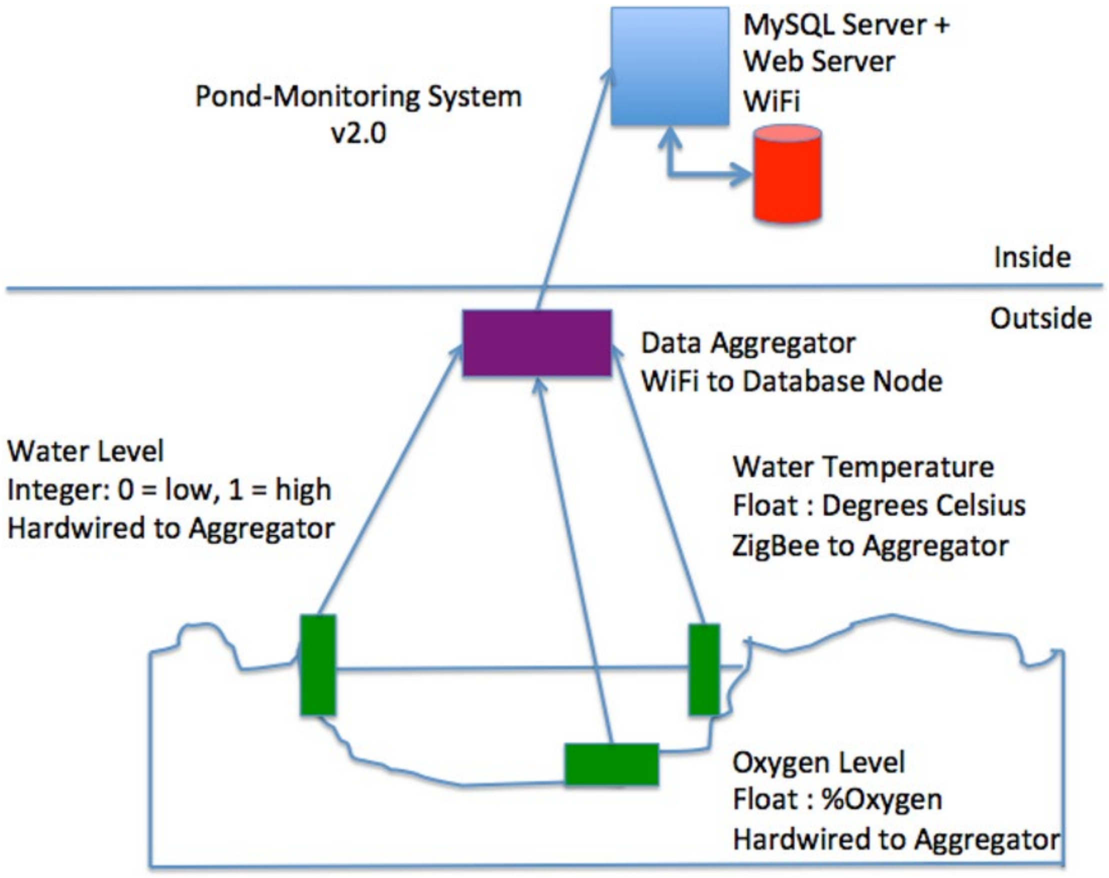

Figure 3-2. *Data flow chart*

This example is for a pond-monitoring system. I show the data collected in its original form and how the data should be interpreted. This helps me design a database to store the data and decide where to place any code or features that may be helpful interpreting the data.

For example, the water-level detection uses an audible tone generated on the Arduino that hosts the sensor. This allows me to hear a warning when the water level drops (the sensor is triggered). Since I walk by the pond every day on my way to and from work, it is a great example of using multiple cues for the user interface. In this case, the user interface is a web page and the signal generator is on the Arduino.

[12]http://uml.org

I also include the communication mechanism. As you can see, two of the data collectors are hardwired to the data aggregator and therefore communicate the data collected via wired connections (I2C in this case). Another data collector uses an XBee (ZigBee protocol) because the water temperature sensor is located too far from the data aggregator (which has to be near the house to transmit data via WiFi to the database server). Finally, notice I use the database server node to host a web server to present the data via a web page.

I recommend using a drawing like this to help you plan your IOT network and even the features for your solution. You really cannot have too much design documentation.

Presentation

When you plan an IOT solution that contains features that permit users to see the data or control actionable devices, you have another level of placement to consider.

This could be rather straightforward if the device is a tablet or computer. In this case, you simply connect it to the same network as the database server and actionable devices. However, if the presentation features are cloud enabled, you may need an intermediate node to isolate your internal nodes from the cloud. This could be a device placed outside the internal firewall, which communicates to the database server and transmits data to a cloud service.

In short, be sure to consider how the data in your database will be presented to the user so that you don't end up with a well-designed data collection mechanism bereft of visualization features. While this sounds obvious, sometimes concentrating on the node placement, data collection, and database design can overshadow how the data will be presented.

Summary

Choosing how to store the data for your IOT solution has many options. You can choose to store the data in the cloud, storing nothing locally on any of the nodes. You could choose to store the data locally in files or memory, building your own storage and retrieval mechanism. Or you could choose to employ a database server dedicated to storing and retrieving your IOT data efficiently and effectively.

Of course, for this book, the assumption is you have chosen or will choose to use a database server to store your data. While you still may choose to cache or even keep a copy locally on some of the nodes, ultimately the database server becomes the focal point for your data.

In this chapter, you examined some of the methods available to you for storing data. You saw examples of how to read and write data in files on the Arduino and Raspberry Pi. I also discussed the benefits, considerations, and recommendations for deploying a database server in your IOT solution. Finally, I discussed some best practices for designing a network of nodes for your IOT solution.

In the next chapter, you'll explore the details of transforming data, from working with data types to normalizing the data, and even talk about how to work with addressing and aggregation. This discussion will prepare you to learn the finer details of working with the MySQL database system.