



# Data Wrangling with JavaScript

Sample Chapter

Ashley Davis





# ***Data Wrangling with JavaScript***

by Ashley Davis

## **Chapter 1**

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

# 1

## *Getting started: establishing your data pipeline*

---

### ***This chapter covers***

- Understanding the what and why of data wrangling
- Defining the difference between data wrangling and data analysis
- Learning when it's appropriate to use JavaScript for data analysis
- Gathering the tools you need in your toolkit for JavaScript data wrangling
- Walking through the data-wrangling process
- Getting an overview of a real data pipeline

### **1.1 Why data wrangling?**

Our modern world seems to revolve around data. You see it almost everywhere you look. If data can be collected, then it's being collected, and sometimes you must try to make sense of it.

Analytics is an essential component of decision-making in business. How are users responding to your app or service? If you make a change to the way you do business, does it help or make things worse? These are the kinds of questions that businesses

are asking of their data. Making better use of your data and getting useful answers can help put us ahead of the competition.

Data is also used by governments to make policies based on evidence, and with more and more *open data* becoming available, citizens also have a part to play in analyzing and understanding this data.

Data wrangling, the act of preparing your data for interrogation, is a skill that's in demand and on the rise. Proficiency in data-related skills is becoming more and more prevalent and is needed by a wider variety of people. In this book you'll work on your data-wrangling skills to help you support data-related activities.

These skills are also useful in your day-to-day development tasks. How is the performance of your app going? Where is the performance bottleneck? Which way is your bug count heading? These kinds of questions are interesting to us as developers, and they can also be answered through data.

## 1.2 **What's data wrangling?**

Wikipedia describes data wrangling as the process of converting data, with the help of tools, from one form to another to allow convenient consumption of the data. This includes transformation, aggregation, visualization, and statistics. I'd say that data wrangling is the whole process of working with data to get it into and through your pipeline, whatever that may be, from data acquisition to your target audience, whoever they might be.

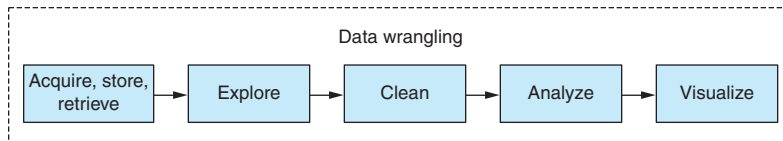
Many books only deal with data analysis, which Wikipedia describes as the process of working with and inspecting data to support decision-making. I view data analysis as a subset of the data-wrangling process. A data analyst might not care about databases, REST APIs, streaming data, real-time analysis, preparing code and data for use in production, and the like. For a data wrangler, these are often essential to the job.

A data analyst might spend most of the time analyzing data offline to produce reports and visualizations to aid decision-makers. A data wrangler also does these things, but they also likely have production concerns: for example, they might need their code to execute in a real-time system with automatic analysis and visualization of live data.

The data-wrangling puzzle can have many pieces. They fit together in many different and complex ways. First, you must acquire data. The data may contain any number of problems that you need to fix. You have many ways you can format and deliver the data to your target audience. In the middle somewhere, you must store the data in an efficient format. You might also have to accept streaming updates and process incoming data in real time.

Ultimately the process of data wrangling is about communication. You need to get your data into a shape that promotes clarity and understanding and enables fast decision-making. How you format and represent the data and the questions you need to ask of it will vary dramatically according to your situation and needs, yet these questions are critical to achieving an outcome.

Through data wrangling, you corral and cajole your data from one shape to another. At times, it will be an extremely messy process, especially when you don't control the



**Figure 1.1** Separating data wrangling into phases

source. In certain situations, you’ll build ad hoc data processing code that will be run only once. This won’t be your best code. It doesn’t have to be because you may never use it again, and you shouldn’t put undue effort into code that you won’t reuse. For this code, you’ll expend only as much effort as necessary to prove that the output is reliable.

At other times, data wrangling, like any coding, can be an extremely disciplined process. You’ll have occasions when you understand the requirements well, and you’ll have patiently built a production-ready data processing pipeline. You’ll put great care and skill into this code because it will be invoked many thousands of times in a production environment. You may have used *test-driven development*, and it’s probably some of the most robust code you’ve ever written.

More than likely your data wrangling will be somewhere within the spectrum between ad hoc and disciplined. It’s likely that you’ll write a bit of throw-away code to transform your source data into something more usable. Then for other code that must run in production, you’ll use much more care.

The process of data wrangling consists of multiple phases, as you can see in figure 1.1. This book divides the process into these phases as though they were distinct, but they’re rarely cleanly separated and don’t necessarily flow neatly one after the other. I separate them here to keep things simple and make things easier to explain. In the real world, it’s never this clean and well defined. The phases of data wrangling intersect and interact with each other and are often tangled up together. Through these phases you understand, analyze, reshape, and transform your data for delivery to your audience.

The main phases of data wrangling are data acquisition, exploration, cleanup, transformation, analysis, and finally reporting and visualization.

Data wrangling involves wrestling with many different issues. How can you filter or optimize data, so you can work with it more effectively? How can you improve your code to process the data more quickly? How do you work with your language to be more effective? How can you scale up and deal with larger data sets?

Throughout this book you’ll look at the process of data wrangling and each of its constituent phases. Along the way we’ll discuss many issues and how you should tackle them.

### 1.3 Why a book on JavaScript data wrangling?

JavaScript isn’t known for its data-wrangling chops. Normally you’re told to go to other languages to work with data. In the past I’ve used Python and Pandas when working with data. That’s what everyone says to use, right? Then why write this book?

Python and Pandas *are* good for data analysis. I won’t attempt to dispute that. They have the maturity and the established ecosystem.

Jupyter Notebook (formerly IPython Notebook) is a great environment for exploratory coding, but you have this type of tool in JavaScript now. Jupyter itself has a plugin that allows it to run JavaScript. Various JavaScript-specific tools are also now available, such as RunKit, Observable, and my own offering is Data-Forge Notebook.

I've used Python for working with data, but I always felt that it didn't fit well into my development pipeline. I'm not saying there's anything wrong with Python; in many ways, I like the language. My problem with Python is that I already do much of my work in JavaScript. I need my data analysis code to run in JavaScript so that it will work in the JavaScript production environment where I need it to run. How do you do that with Python?

You could do your exploratory and analysis coding in Python and then move the data to JavaScript visualization, as many people do. That's a common approach due to JavaScript's strong visualization ecosystem. But then what if you want to run your analysis code on live data? When I found that I needed to run my data analysis code in production, I then had to rewrite it in JavaScript. I was never able to accept that this was the way things must be. For me, it boils down to this: I don't have time to rewrite code.

But does anyone have time to rewrite code? The world moves too quickly for that. We all have deadlines to meet. You need to add value to your business, and time is a luxury you can't often afford in a hectic and fast-paced business environment. You want to write your data analysis code in an exploratory fashion, à la Jupyter Notebook, but using JavaScript and later deploying it to a JavaScript web application or microservice.

This led me on a journey of working with data in JavaScript and building out an open source library, Data-Forge, to help make this possible. Along the way I discovered that the data analysis needs of JavaScript programmers were not well met. This state of affairs was somewhat perplexing given the proliferation of JavaScript programmers, the easy access of the JavaScript language, and the seemingly endless array of JavaScript visualization libraries. Why weren't we already talking about this? Did people really think that data analysis couldn't be done in JavaScript?

These are the questions that led me to write this book. If you know JavaScript, and that's the assumption I'm making, then you probably won't be surprised that I found JavaScript to be a surprisingly capable language that gives substantial productivity. For sure, it has problems to be aware of, but all good JavaScript coders are already working with the good parts of the language and avoiding the bad parts.

These days all sorts of complex applications are being written in JavaScript. You already know the language, it's capable, and you use it in production. Staying in JavaScript is going to save you time and effort. Why not also use JavaScript for data wrangling?

## **1.4 What will you get out of this book?**

You'll learn how to do data wrangling in JavaScript. Through numerous examples, building up from simple to more complex, you'll develop your skills for working with data. Along the way you'll gain an understanding of the many tools you can use that are

already readily available to you. You'll learn how to apply data analysis techniques in JavaScript that are commonly used in other languages.

Together we'll look at the entire data-wrangling process purely in JavaScript. You'll learn to build a data processing pipeline that takes the data from a source, processes and transforms it, then finally delivers the data to your audience in an appropriate form.

You'll learn how to tackle the issues involved in rolling out your data pipeline to your production environment and scaling it up to large data sets. We'll look at the problems that you might encounter and learn the thought processes you must adopt to find solutions.

I'll show that there's no need for you to step out to other languages, such as Python, that are traditionally considered better suited to data analysis. You'll learn how to do it in JavaScript.

The ultimate takeaway is an appreciation of the world of data wrangling and how it intersects with JavaScript. This is a huge world, but *Data Wrangling with JavaScript* will help you navigate it and make sense of it.

## 1.5 Why use JavaScript for data wrangling?

I advocate using JavaScript for data wrangling for several reasons; these are summarized in table 1.1.

**Table 1.1** Reasons for using JavaScript for data wrangling

Reason	Details
You already know JavaScript.	Why learn another language for working with data? (Assuming you already know JavaScript.)
JavaScript is a capable language.	It's used to build all manner of complex applications.
Exploratory coding.	Using a prototyping process with live reload (discussed in chapter 5) is a powerful way to write applications using JavaScript.
Strong visualization ecosystem.	Python programmers often end up in JavaScript to use its many visualization libraries, including D3, possibly the most sophisticated visualization library. We'll explore visualization in chapters 10 and 13.
Generally strong ecosystem.	JavaScript has one of the strongest user-driven ecosystems. Throughout the book we'll use many third-party tools, and I encourage you to explore further to build out your own toolkit.
JavaScript is everywhere.	JavaScript is in the browser, on the server, on the desktop, on mobile devices, and even on embedded devices.
JavaScript is easy to learn.	JavaScript is renowned for being easy to get started with. Perhaps it's hard to master, but that's also true of any programming language.



**Table 1.1** Reasons for using JavaScript for data wrangling (*continued*)

Reason	Details
JavaScript programmers are easy to find.	In case you need to hire someone, JavaScript programmers are everywhere.
JavaScript is evolving.	The language continues to get safer, more reliable, and more convenient. It's refined with each successive version of the ECMAScript standard.
JavaScript and JSON go hand in hand.	The JSON data format, the data format of the web, evolved from JavaScript. JavaScript has built-in tools for working with JSON as do many third-party tools and libraries.

## 1.6 *Is JavaScript appropriate for data analysis?*

We have no reason to single out JavaScript as a language that's *not* suited to data analysis. The best argument against JavaScript is that languages such as Python or R, let's say, have more *experience* behind them. By this, I mean they've built up a reputation and an ecosystem for this kind of work. JavaScript can get there as well, if that's how you want to use JavaScript. It certainly is how I want to use JavaScript, and I think once data analysis in JavaScript takes off it will move quickly.

I expect criticism against JavaScript for data analysis. One argument will be that JavaScript doesn't have the performance. Similar to Python, JavaScript is an interpreted language, and both have restricted performance because of this. Python works around this with its well-known native C libraries that compensate for its performance issues. Let it be known that JavaScript has native libraries like this as well! And while JavaScript was never the most high-performance language in town, its performance has improved significantly thanks to the innovation and effort that went into the V8 engine and the Chrome browser.

Another argument against JavaScript may be that it isn't a high-quality language. The JavaScript language has design flaws (what language doesn't?) and a checkered history. As JavaScript coders, you've learned to work around the problems it throws at us, and yet you're still productive. Over time and through various revisions, the language continues to evolve, improve, and become a better language. These days I spend more time with *TypeScript* than JavaScript. This provides the benefits of *type safety* and *intellisense* when needed, on top of everything else to love about JavaScript.

One major strength that Python has in its corner is the fantastic exploratory coding environment that's now called Jupyter Notebook. Please be aware, though, that Jupyter now works with JavaScript! That's right, you can do exploratory coding in Jupyter with JavaScript in much the same way professional data analysts use Jupyter and Python. It's still early days for this . . . it does work, and you can use it, but the experience is not yet as complete and polished as you'd like it.

Python and R have strong and established communities and ecosystems relating to data analysis. JavaScript also has a strong community and ecosystem, although it doesn't

yet have that strength in the area of data analysis. JavaScript *does* have a strong data visualization community and ecosystem. That's a great start! It means that the output of data analysis often ends up being visualized in JavaScript anyway. Books on bridging Python to JavaScript attest to this, but working across languages in that way sounds inconvenient to me.

JavaScript will never take away the role for Python and R for data analysis. They're already well established for data analysis, and I don't expect that JavaScript could ever overtake them. Indeed, it's not my intention to turn people away from those languages. I would, however, like to show JavaScript programmers that it's possible for them to do everything they need to do without leaving JavaScript.

## 1.7 Navigating the JavaScript ecosystem

The JavaScript ecosystem is huge and can be overwhelming for newcomers. Experienced JavaScript developers treat the ecosystem as part of their toolkit. Need to accomplish something? A package that does what you want on npm (node package manager) or Bower (client-side package manager) probably already exists.

Did you find a package that almost does what you need, but not quite? Most packages are open source. Consider forking the package and making the changes you need.

Many JavaScript libraries will help you in your data wrangling. At the start of writing, npm listed 71 results for *data analysis*. This number has now grown to 115 as I near completion of this book. There might already be a library there that meets your needs.

You'll find many tools and frameworks for visualization, building user interfaces, creating dashboards, and constructing applications. Popular libraries such as Backbone, React, and AngularJS come to mind. These are useful for building web apps. If you're creating a build or automation script, you'll probably want to look at Grunt, Gulp, or Task-Mule. Or search for *task runner* in npm and choose something that makes sense for you.

## 1.8 Assembling your toolkit

As you learn to be data wranglers, you'll assemble your toolkit. Every developer needs tools to do the job, and continuously upgrading your toolkit is a core theme of this book. My most important advice to any developer is to make sure that you have good tools and that you know how to use them. Your tools must be reliable, they must help you be productive, and you must understand how to use them well.

Although this book will introduce you to many new tools and techniques, we aren't going to spend any time on fundamental development tools. I'll take it for granted that you already have a text editor and a version control system and that you know how to use them.

For most of this book, you'll use Node.js to develop code, although most of the code you write will also work in the browser, on a mobile (using Ionic), or on a desktop (using Electron). To follow along with the book, you should have Node.js installed. Packages and dependencies used in this book can be installed using npm, which comes with

Node.js or with Bower that can be installed using npm. Please read chapter 2 for help coming up to speed with Node.js.

You likely already have a favorite testing framework. This book doesn't cover automated unit or integration testing, but please be aware that I do this for my most important code, and I consider it an important part of my general coding practice. I currently use Mocha with Chai for JavaScript unit and integration testing, although there are other good testing frameworks available. The final chapter covers a testing technique that I call *output testing*; this is a simple and effective means of testing your code when you work with data.

For any serious coding, you'll already have a method of building and deploying your code. Technically JavaScript doesn't need a build process, but it can be useful or necessary depending on your target environment; for example, I often work with TypeScript and use a build process to compile the code to JavaScript. If you're deploying your code to a server in the cloud, you'll most certainly want a provisioning and deployment script. Build and deployment aren't a focus of this book, but we discuss them briefly in chapter 14. Otherwise I'll assume you already have a way to get your code into your target environment or that's a problem you'll solve later.

Many useful libraries will help in your day-to-day coding. Underscore and Lodash come to mind. The ubiquitous JQuery seems to be going out of fashion at the moment, although it still contains many useful functions. For working with collections of data *linq*, a port of Microsoft LINQ from the C# language, is useful. My own Data-Forge library is a powerful tool for working with data. Moment.js is essential for working with date and time in JavaScript. Cheerio is a library for scraping data from HTML. There are numerous libraries for data visualization, including but not limited to D3, Google Charts, Highcharts, and Flot. Libraries that are useful for data analysis and statistics include jStat, Mathjs, and Formulajs. I'll expand more on the various libraries through this book.

Asynchronous coding deserves a special mention. *Promises* are an expressive and cohesive way of managing your asynchronous coding, and I definitely think you should understand how to use them. Please see chapter 2 for an overview of asynchronous coding and promises.

Most important for your work is having a good setup for exploratory coding. This process is important for inspecting, analyzing, and understanding your data. It's often called *prototyping*. It's the process of rapidly building up code step by step in an iterative fashion, starting from simple beginnings and building up to more complex code—a process we'll use often throughout this book. While prototyping the code, we also delve deep into your data to understand its structure and shape. We'll talk more about this in chapter 5.

In the next section, we'll talk about the data-wrangling process and flesh out a data pipeline that will help you understand how to fit together all the pieces of the puzzle.

## **1.9** *Establishing your data pipeline*

The remainder of chapter 1 is an overview of the data-wrangling process. By the end you'll cover an example of a data processing pipeline for a project. This is a whirlwind

tour of data wrangling from start to end. Please note that this isn't intended to be an example of a typical data-wrangling project—that would be difficult because they all have their own unique aspects. I want to give you a taste of what's involved and what you'll learn from this book.

You have no code examples yet; there's plenty of time for that through the rest of the book, which is full of working code examples that you can try for yourself. Here we seek to understand an example of the data-wrangling process and set the stage for the rest of the book. Later I'll explain each aspect of data wrangling in more depth.

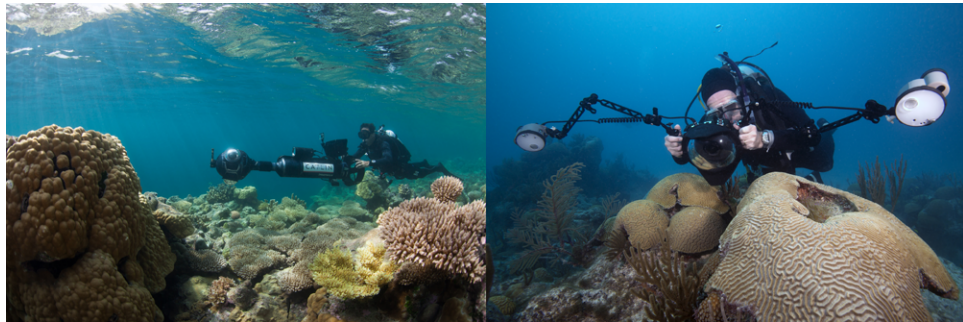
### 1.9.1 Setting the stage

I've been kindly granted permission to use an interesting data set. For various examples in the book, we'll use data from "XL Catlin Global Reef Record." We must thank the University of Queensland for allowing access to this data. I have no connection with the Global Reef Record project besides an interest in using the data for examples in this book.

The reef data was collected by divers in survey teams on reefs around the world. As the divers move along their *survey* route (called a *transect* in the data), their cameras automatically take photos and their sensors take readings (see figure 1.2). The reef and its health are being mapped out through this data. In the future, the data collection process will begin again and allow scientists to compare the health of reefs between then and now.

The reef data set makes for a compelling sample project. It contains time-related data, geo-located data, data acquired by underwater sensors, photographs, and then data generated from images by machine learning. This is a large data set, and for this project I extract and process the parts of it that I need to create a dashboard with visualizations of the data. For more information on the reef survey project, please watch the video at <https://www.youtube.com/watch?v=LBmrBOVMm5Q>.

I needed to build a dashboard with tables, maps, and graphs to visualize and explore the reef data. Together we'll work through an overview of this process, and I'll explain it from beginning to end, starting with capturing the data from the original MySQL



© The Ocean Agency / XL Catlin Seaview Survey / Christophe Bailhache and Jayne Jenkins.

**Figure 1.2** Divers taking measurements on the reef.

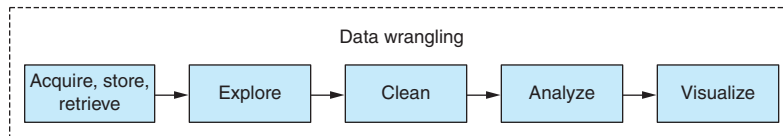
database, processing that data, and culminating in a web dashboard to display the data. In this chapter, we take a bird's-eye view and don't dive into detail; however, in later chapters we'll expand on various aspects of the process presented here.

Initially I was given a sample of the reef data in CSV (comma-separated value) files. I explored the CSV for an initial understanding of the data set. Later I was given access to the full MySQL database. The aim was to bring this data into a production system. I needed to organize and process the data for use in a real web application with an operational REST API that feeds data to the dashboard.

### 1.9.2 The data-wrangling process

Let's examine the data-wrangling process: it's composed of a series of phases as shown in figure 1.3. Through this process you acquire your data, explore it, understand it, and visualize it. We finish with the data in a production-ready format, such as a web visualization or a report.

Figure 1.3 gives us the notion that this is a straightforward and linear process, but if you have previous experience in software development, you'll probably smell a rat here. Software development is rarely this straightforward, and the phases aren't usually cleanly separated, so don't be too concerned about the order of the phases presented here. I have to present them in an order that makes sense, and a linear order is a useful structure for the book. In chapter 5 you'll move beyond the linear model of software development and look at an iterative *exploratory* model.



**Figure 1.3** The data-wrangling process

As you work through the process in this chapter, please consider that this isn't *the process*; rather this is an example of what the data-wrangling process looks like for a particular project. How the process manifests itself will be different depending on your data and requirements. When you embark on other projects, your own process will undoubtedly look different than what I describe in this chapter.

### 1.9.3 Planning

Before getting into data wrangling, or any project for that matter, you should understand what you're doing. What are your requirements? What and how are you going to build your software? What problems are likely to come up, and how will you deal with them? What does your data look like? What questions should you ask of the data? These are the kinds of questions you should ask yourself when planning a new project.

When you're doing any sort of software development, it's important to start with planning. The biggest problem I see in many programmers is their failure to think and plan out their work before coding. In my experience, one of the best ways to improve as a coder is to become better at planning.

Why? Because planning leads to better outcomes through better implementation and fewer mistakes. But you must be careful not to *over* plan! Planning for a future that's unlikely to happen leads to overengineering.

You might need to do *exploratory coding* before you can plan! This is an example of the phases not being cleanly separated. If you don't have enough information to plan, then move forward with exploratory coding and return to planning when you have a better understanding of the problem you're trying to solve.

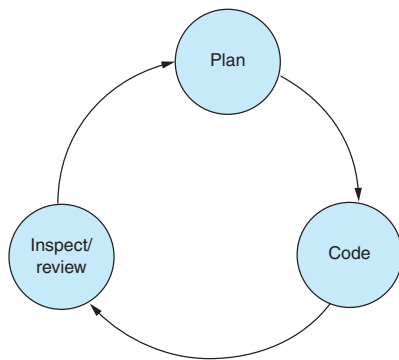
Planning is an important part of an effective feedback loop (see figure 1.4). Planning involves working through the mistakes that will likely happen and figuring out how to avoid those mistakes. Avoiding mistakes saves you much time and anguish. Each trip around the feedback loop is a valuable experience, improving your understanding of the project and your ability to plan and execute.

To plan this project, let's note several requirements for the end product:

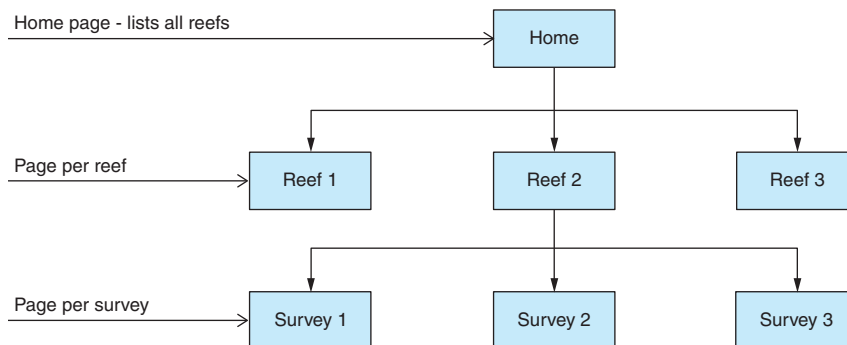
- Create a web dashboard to provide easy browsing of the reef data.
- Summarize reefs and surveys completed through tables, charts, and maps.

Requirements usually change over time as you develop your understanding of the project. Don't be concerned if this happens. Changing requirements is natural, but be careful: it can also be symptomatic of poor planning or scope creep.

At this stage, I plan the structure of the website, as shown in the figure 1.5.



**Figure 1.4** The feedback loop



**Figure 1.5** Dashboard website structure

Simple wireframe mockups can help us solidify the plan. Figure 1.6 is an example. During planning, you need to think of the problems that might arise. This will help you to preemptively plan solutions to those problems, but please make sure your approach is balanced. If you believe a problem has little chance of arising, you should spend little effort mitigating against it. For example, here are several of the problems that I might encounter while working with the reef data set and building the dashboard:

- Due to its size, several of the tables contain more than a million records. It might take a long time to copy the MySQL database, although it can run for as many hours as we need it to. I have little need to optimize this process because it happens only once, so it isn't time critical.
- There will likely be problems with the data that need to be cleaned up, but I won't know about those until I explore the data set (see chapter 6 for data cleanup and preparation).
- If the visualizations in the dashboard are slow to load or sluggish in performance, you can prebake the data into an optimized format (see chapters 6 and 7 for more on this).

Of primary importance in the planning phase is to have an idea of what you want from the data. Ask yourself the following questions: What do you need to know from the data? What questions are you asking of the data?

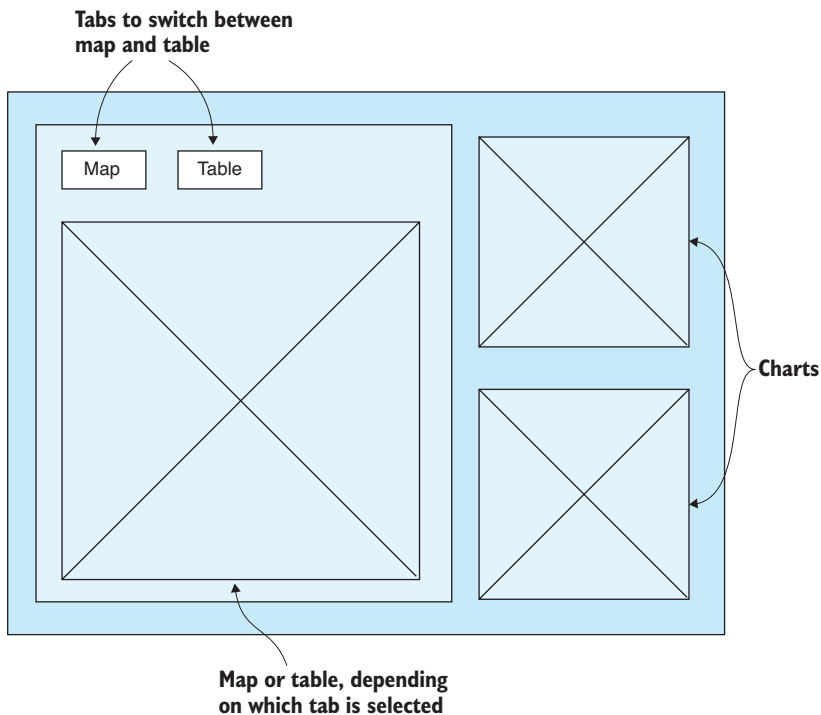


Figure 1.6 Dashboard page mockup



For your example, here are several of the questions to ask of the reef data:

- What's the average temperature per reef in Australia reefs that were surveyed?
- What's the total coverage (distance traversed) for each reef?
- What's the average dive depth per reef?

Often, despite planning, you may find that things don't go according to plan. When this happens, take a break and take time to reassess the situation. When necessary, come back to planning and work through it again. Return to planning at any time when things go wrong or if you need confirmation that you're on the right track.

### 1.9.4 Acquisition, storage, and retrieval

In this phase, you capture the data and store it in an appropriate format. You need the data stored in a format where you can conveniently and effectively query and retrieve it.

Data acquisition started with a sample CSV file that was emailed from the University of Queensland. I did a *mini exploration* of the sample data to get a feel for it. The sample data was small enough that I could load it in Excel.

I needed to get an idea of what I was dealing with before writing any code. When looking at the full data set, I used a SQL database viewer called HeidiSQL (figure 1.7) to connect to the remote database, explore the data, and develop understanding of it.

Due to slow internet speeds, remote data access wasn't going to work well for exploratory coding. I needed to download the data to a local database for efficient access. I also wanted the data locally so that I could make changes to it as needed, and I couldn't make changes to a database that I didn't own. I planned to copy the data down to a local MongoDB database (figure 1.8).

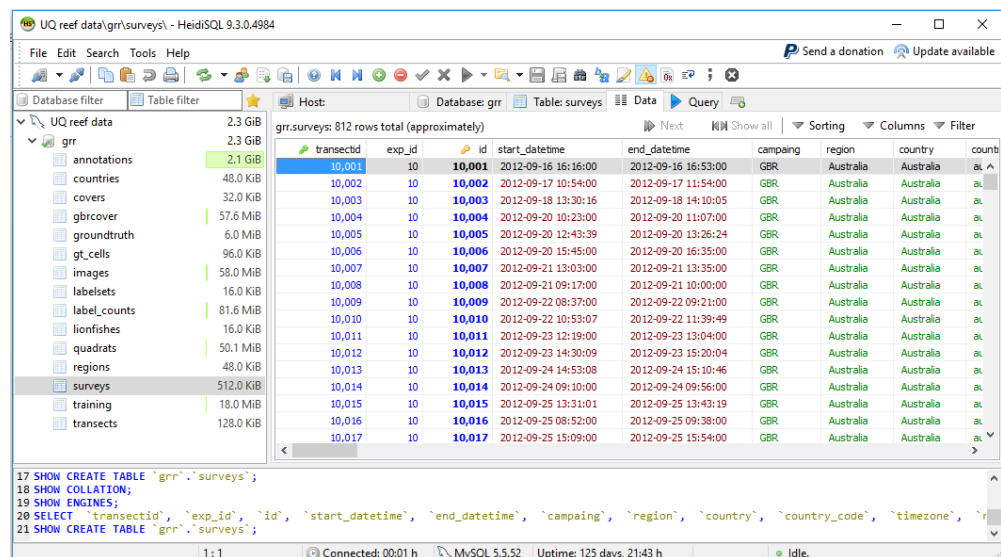
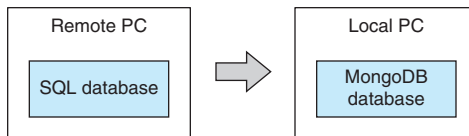


Figure 1.7 Inspecting an SQL table in HeidiSQL





**Figure 1.8** Pulling the data from SQL to MongoDB

You might wonder why I chose MongoDB? Well, the choice is somewhat arbitrary. You need to choose a database that works well for you and your project. I like MongoDB for several reasons:

- It's simple to install.
- It works well with JavaScript and JSON.
- It's easy to store and retrieve data.
- The query language is built into the programming language.
- Ad hoc or irregular data can be stored.
- It has good performance.

If you're concerned that moving the data from SQL to MongoDB will cause the data to lose structure, please don't be: MongoDB can store structured and relational data just as well as SQL. They're different, and MongoDB doesn't have the convenience of SQL *joins* and it doesn't *enforce* structure or relationships—but these are features that you can easily emulate in your own code.

Something else that's important with MongoDB is that there's no need to predefine a schema. You don't have to commit to the final shape of your data! That's great because I don't yet know the final shape of my data. Not using a schema reduces the burden of designing your data, and it allows you to more easily evolve your data as you come to understand your project better.

You'll learn more about SQL, MongoDB, and other data sources in chapter 3.

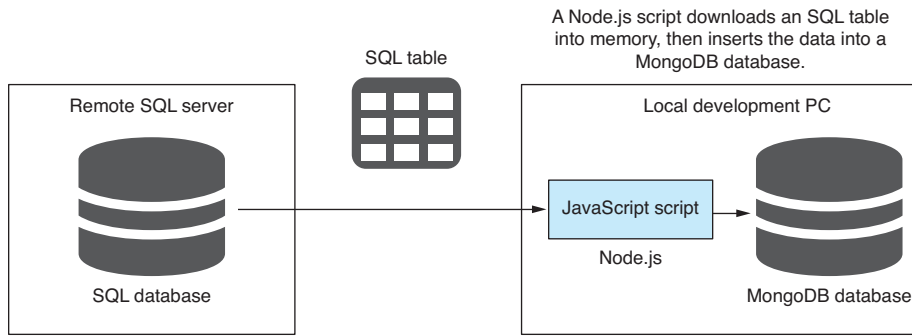
At this point it's time to start coding. I must write a script to copy from the SQL database to MongoDB. I start by using `nodejs-mysql` to load a MySQL table into memory from the remote database. With large databases, this isn't realistic, but it did work on this occasion. In chapters 8 and 9, we'll talk about working with data sets that are too large to fit into memory.

With the SQL table loaded into memory, you now use the MongoDB API to insert the data into our local MongoDB database instance (figure 1.9).

Now I can assemble the code I have so far, and I have a Node.js script that can replicate a MySQL table to MongoDB. I can now easily scale this up and have a script that can replicate the entire MySQL database to our local MongoDB instance.

How much data am I pulling down and how long will it take? Note here that I'm not yet processing the data or transforming it in any way. That comes later when I have a local database and a better understanding of the data.

It took many hours to replicate this database, and that's with a lousy internet connection. Long-running processes like this that depend on fragile external resources should be designed to be fault-tolerant and restartable. We'll touch on these points again in chapter 14. The important thing, though, is that most of the time the script was doing its work without intervention, and it didn't *cost* much of my own time. I'm happy to wait



**Figure 1.9** Downloading an SQL database table with a Node.js script

for this process to complete because having a local copy of the data makes all future interactions with it more efficient.

Now that I have a local copy of the database, we are almost ready to begin a more complete exploration of the data. First, though, I must retrieve the data.

I use the MongoDB API to query the local database. Unlike SQL, the MongoDB query language is integrated into JavaScript (or other languages, depending on your language of choice).

In this case, you can get away with a basic query, but you can do so much more with a MongoDB query, including

- Filtering records
- Filtering data returned for each record
- Sorting records
- Skipping and limiting records to view a reduced *window* of the data

This is one way to acquire data, but many other ways exist. Many different data formats and data storage solutions can be used. You'll dive into details on MongoDB in chapter 8.

### 1.9.5 Exploratory coding

In this phase, you use code to deeply explore your data and build your understanding of it. With a better understanding, you can start to make assumptions about the structure and consistency of the data. Assumptions must be checked, but you can do that easily with code!

We write code to poke, prod, and tease the data. We call this *exploratory coding* (also often called *prototyping*), and it helps us get to know our data while producing potentially useful code.

It's important to work with a smaller subset of data at this point. Attempting to work with the entire data set can be inefficient and counterproductive, although of course it depends on the size of your particular data set.

Exploratory coding is the process of incrementally building your code through an iterative and interactive process (figure 1.10). Code a few lines, then run the code and inspect the output, repeat. Repeating this process builds up your code and understanding at the same time.

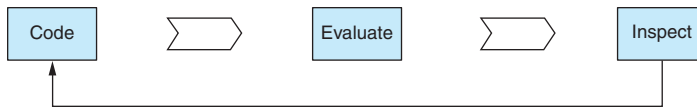


Figure 1.10 Exploratory coding process

The simplest way to start looking at the data is to use a database viewer. I already used HeidiSQL to look at the SQL database. Now I use Robomongo (recently renamed to Robo 3T) to look at the contents of my local MongoDB database (figure 1.11).

Using code, I explore the data, looking at the first and last records and the data types they contain. I print the first few records to the console and see the following:

```
> [ { _id: 10001,
    reef_name: 'North Opal Reef',
    sub_region: 'Cairns-Cooktown',
    local_region: 'Great Barrier Reef',
    country: 'Australia',
    region: 'Australia',
    latitude: -16.194318893060213,
    longitude: 145.89624754492613 },
  { _id: 10002,
    reef_name: 'North Opal Reef',
    sub_region: 'Cairns-Cooktown',
    local_region: 'Great Barrier Reef',
    country: 'Australia',
    region: 'Australia',
    latitude: -16.18198943421998,
    longitude: 145.89718533957503 },
  { _id: 10003,
    reef_name: 'North Opal Reef',
    sub_region: 'Cairns-Cooktown',
    local_region: 'Great Barrier Reef',
    country: 'Australia',
    region: 'Australia',
    latitude: -16.17732916639253,
    longitude: 145.88907464416826 } ]
```

Each column is a field in the document.

Each row is a document in the collection.

	_id	reef_name	sub_region	local_region	country	region	latitude	longitude
1	10001	North Opal Reef	Cairns-Cooktown	Great Barrier Reef	Australia	Australia	-16.194318893060213	145.89624754492613
2	10002	North Opal Reef	Cairns-Cooktown	Great Barrier Reef	Australia	Australia	-16.18198943421998	145.89718533957503
3	10003	North Opal Reef	Cairns-Cooktown	Great Barrier Reef	Australia	Australia	-16.17732916639253	145.88907464416826
4	10004	Holmes Reef	Coral Sea Commonwealth Marine Reserve	Coral Sea	Australia	Australia	-16.530294027699	147.823509740939
5	10005	Holmes Reef	Coral Sea Commonwealth Marine Reserve	Coral Sea	Australia	Australia	-16.5237831228651	147.82483822933
6	10006	Holmes Reef	Coral Sea Commonwealth Marine Reserve	Coral Sea	Australia	Australia	-16.5141587263537	147.843373952857
7	10007	Holmes Reef	Coral Sea Commonwealth Marine Reserve	Coral Sea	Australia	Australia	-16.38519722313	147.850103887981
8	10008	Holmes Reef	Coral Sea Commonwealth Marine Reserve	Coral Sea	Australia	Australia	-16.4397562219759	147.900277651823
9	10009	Holmes Reef	Coral Sea Commonwealth Marine Reserve	Coral Sea	Australia	Australia	-16.4034044932481	148.016757044630
10	10010	Holmes Reef	Coral Sea Commonwealth Marine Reserve	Coral Sea	Australia	Australia	-16.4109380274621	148.037110698863
11	10011	Flinders Reef	Coral Sea Commonwealth Marine Reserve	Coral Sea	Australia	Australia	-17.712454885141	148.392627912528
12	10012	Flinders Reef	Coral Sea Commonwealth Marine Reserve	Coral Sea	Australia	Australia	-17.7136335641385	148.351670936661

Figure 1.11 Looking at the transects collection in Robomongo

From looking at the data, I'm getting a feel for the shape of it and can ask the following questions: What columns do I have? How many records am I dealing with? Again, using code, I analyze the data and print the answers to the console:

```
Num columns: 59
Columns:      _id, transectid, exp_id, start_datetime, ...
Num records: 812
```

With the help of my open source data-wrangling toolkit Data-Forge, I can understand the types of data and the frequency of the values. I print the results to the console and learn even more about my data:

__index__	Type	Frequency	Column
0	number	100	_id
1	number	100	transectid
2	number	100	exp_id
3	string	100	start_datetime
4	string	100	end_datetime
5	string	100	campaing
...			

__index__	Value	Frequency	Column
0	Australia	31.896551724137932	region
1	Atlantic	28.57142857142857	region
2	Southeast Asia	16.133004926108374	region
3	Pacific	15.024630541871922	region
...			

You'll learn more about using Data-Forge and what it can do throughout the book, especially in chapter 9.

Now that I have a basic understanding of the data, I can start to lay out our assumptions about it. Is each column expected to have only a certain type of data? Is the data consistent?

Well, I can't know this yet. I'm working with a large data set, and I haven't yet looked at every single record. In fact, I can't manually inspect each record because I have too many! However, I can easily use code to test my assumptions.

I write an *assumption checking* script that will verify my assumptions about the data. This is a Node.js script that inspects each record in the database and checks that each field contains values with the same types that we expect. You'll look at code examples for assumption checking in chapter 5.

Data can sometimes be frustratingly inconsistent. Problems can easily hide for a long time in large data sets. My assumption checking script gives me peace of mind and reduces the likelihood that I'll later be taken by surprise by nasty issues in the data.

Running the assumption checking script shows that my assumptions about the data don't bear out. I find that I have unexpected values in the `dive_temperature` field that I can now find on closer inspection in Robomongo (figure 1.12).

Why is the data broken? That's hard to say. Maybe several of the sensors were faulty or working intermittently. It can be difficult to understand why faulty data comes into your system the way it does.



ve_maximum_d	dive_visibility	dive_temperature
12.6	20	0
13.5	20	0
16.6	20	25.5
12	0	0
13.3	40	26.4
14	0	0
14.6	0	26.2
13.1	0	0
12.6	40	26.3
16.3	40	26.5
14.7	30	26.1
12.1	30	26.1
12	30	0
12	0	25
10.5	0	26.5

**Figure 1.12** Inspecting bad temperature values in Robomongo

What if the data doesn't meet expectations? Then we have to rectify the data or adapt our workflow to fit, so next we move on to data cleanup and preparation.

You've finished this section, but you haven't yet finished your exploratory coding. You can continue exploratory coding throughout all phases of data wrangling. Whenever you need to try something new with the data, test an idea, or test code, you can return to exploratory coding to iterate and experiment. You'll spend a whole chapter on exploratory coding in chapter 5.

### 1.9.6 *Clean and prepare*

Did your data come in the format you expected? Is your data fit for production usage? In the *clean and prepare* phase, you rectify issues with the data and make it easier to deal with downstream. You can also normalize it and restructure it for more efficient use in production.

The data you receive might come in any format! It might contain any number of problems. It doesn't matter; you still have to deal with it. The assumption checking script has already found that the data isn't willing to conform to my expectations! I have work to do now to clean up the data to make it match my desired format.

I know that my data contains invalid temperature values. I could remove records with invalid temperatures from my database, but then I'd lose other useful data. Instead, I'll work around this problem later, filtering out records with invalid temperatures as needed.

For the sake of an example, let's look at a different problem: the date/time fields in the *surveys* collection. You can see that this field is stored as a string rather than a JavaScript date/time object (figure 1.13).

With date/time fields stored as strings, this opens the possibility that they might be stored with inconsistent formats. In reality, my sample data is well structured in this regard, but let's imagine for this example that several of the dates are stored with time zone information that assume an Australian time zone. This sort of thing can be an insidious and well-hidden problem; working with dates/times often has difficulties like this.

```

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

/* 1 */
{
  "_id" : 10001,
  "transectid" : 10001,
  "exp_id" : 10,
  "start_datetime" : "2012-09-16 16:16:00",
  "end_datetime" : "2012-09-16 16:53:00",
  "campaign" : "GBR",
  "region" : "Australia",
  "country" : "Australia",
  "country_code" : "au",
  "timezone" : 10,
  "reef_name" : "Opal Reef",
  "reef_type" : "Outer",
  "sub_region" : "Cairns-Cooktown",
}

```

Figure 1.13 Date/time fields in the surveys collection are string values.

To fix this data, I write another Node.js script. For each record, it examines the fields and if necessary fixes the data. It must then save the repaired data back to the database. This kind of issue isn't difficult to fix; it's spotting the problem in the first place that's the difficult part. But you might also stumble on other issues that aren't so easy to fix, and fixing them could be time consuming. In many cases, it will be more efficient to deal with the bad data at runtime rather than trying to fix it offline.

At this stage, you might also consider normalizing or standardizing your data to ensure that it's in a suitable format for analysis, to simplify your downstream code, or for better performance. We'll see more examples of data problems and fixes in chapter 6.

## 1.9.7 Analysis

In this phase, you analyze the data. You ask and answer specific questions about the data. It's a further step in understanding the data and extrapolating meaningful insights from it.

Now that I have data that's cleaned and prepared for use, it's time to do analysis. I want to do much with the data. I want to understand the total distance traversed in each survey. I want to compute the average water temperature for each reef. I want to understand the average depth for each reef.

I start by looking at the total distance traveled by divers for each reef. I need to aggregate and summarize the data. The aggregation takes the form of grouping by reef. The summarization comes in the form of summing the distance traveled for each reef. Here's the result of this analysis:

__index__	reef_name	distance
-----	-----	-----
Opal Reef	Opal Reef	15.526000000000002
Holmes Reef	Holmes Reef	13.031
Flinders Reef	Flinders Reef	16.344
Myrmidon Reef	Myrmidon Reef	7.263999999999999
Davies Reef	Davies Reef	3.297
...		

The code for this can easily be extended. For example, I already have the data grouped by reef, so I'll add average temperature per reef, and now I have both total distance and average temperature:

<code>__index__</code>	<code>reef_name</code>	<code>distance</code>	<code>temperature</code>
Opal Reef	Opal Reef	15.526000000000002	22.625
Holmes Reef	Holmes Reef	13.031	16.487499999999997
Flinders Reef	Flinders Reef	16.344	16.60909090909091
Myrmidon Reef	Myrmidon Reef	7.263999999999999	0
...			

With slight changes to the code I can ask similar questions, such as what's the average temperature by country. This time, instead of grouping by reef, I group by country, which is a different way of looking at the data:

<code>__index__</code>	<code>country</code>	<code>distance</code>
Australia	Australia	350.45000000000004
Curacao	Curacao	38.481000000000001
Bonaire	Bonaire	32.391000000000001
Aruba	Aruba	8.491
Belize	Belize	38.459000000000001

This gives you a taste for data analysis, but stay tuned; you'll spend more time on this and look at code examples in chapter 9.

### 1.9.8 Visualization

Now you come to what's arguably the most exciting phase. Here you visualize the data and bring it to life. This is the final phase in understanding your data. Rendering the data in a visual way can bring forth insights that were otherwise difficult to see.

After you explore and analyze the data, it's time to visualize it and understand it in a different light. Visualization completes your understanding of the data and allows you to easily see what might have otherwise remained hidden. You seek to expose any remaining problems in the data through visualization.

For this section, I need a more complex infrastructure (see figure 1.14). I need

- A server
- A REST API to expose your data
- A simple web application to render the visualization

I build a simple web server using Express.js. The web server hosts a REST API that exposes the reef data using HTTP GET. The REST API is the interface between the server and your web application (figure 1.14).



**Figure 1.14** Infrastructure for a web app with a chart

Next, I create a simple web application that uses the REST API to retrieve the data in JSON format. My simple web app retrieves data from the database using the REST API, and I can put that data to work. I'm using C3 here to render a chart. I add the chart to the web page and use JavaScript to inject the data. We'll learn more about C3 later in the book.

But I have a big problem with the first iteration of the chart. It displays the temperature for each survey, but there's too much data to be represented in a bar chart. And this isn't what I wanted anyway. Instead, I want to show average temperature for each reef, so I need to take the code that was developed in the analysis phase and move that code to the browser. In addition, I filter down the data to reefs in Australia, which helps cut down the data somewhat.

Building on the code from the analysis phase, I filter out non-Australian reefs, group by reef name, and then compute the average temperature for each reef. We then plug this data into the chart. You can see the result in figure 1.15. (To see the color, refer to the electronic versions of the book.)

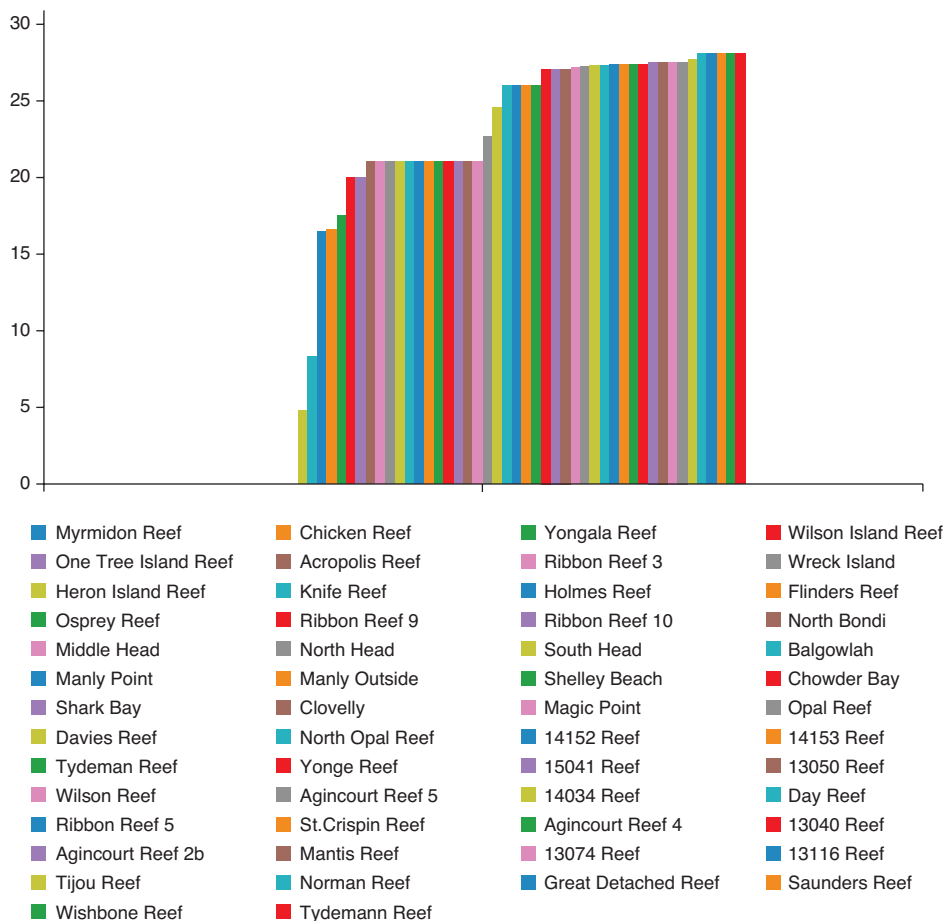


Figure 1.15 Chart showing temperature of reefs in Australia



### 1.9.9 Getting to production

In this final phase of data wrangling, you deliver your data pipeline to your audience. We'll deploy the web app to the *production environment*. This is arguably the most difficult part of this process: bringing a production system online. By production, I mean a system that's in operation and being used by someone, typically a client or the general public. That's where it must exist to reach your audience.

There will be times when you do a one-time data analysis and then throw away the code. When that's adequate for the job, you don't need to move that code to production, so you won't have the concerns and difficulties of such (lucky you), although most of the time you need to get your code to the place where it needs to run.

You might move your code to a web service, a front end, a mobile app, or a desktop app. After moving your code to production, it will run automatically or on demand. Often it will process data in real-time, and it might generate reports and visualizations or whatever it needs to do.

In this case I built a dashboard to display and explore the reef data. The final dashboard looks like figure 1.16.

The code covered so far in this chapter is already in JavaScript, so it isn't difficult to slot it into place in my JavaScript production environment. This is one of the major benefits of doing all our data-related work in JavaScript. As you move through the exploratory phase and toward production, you'll naturally take more care with your coding.

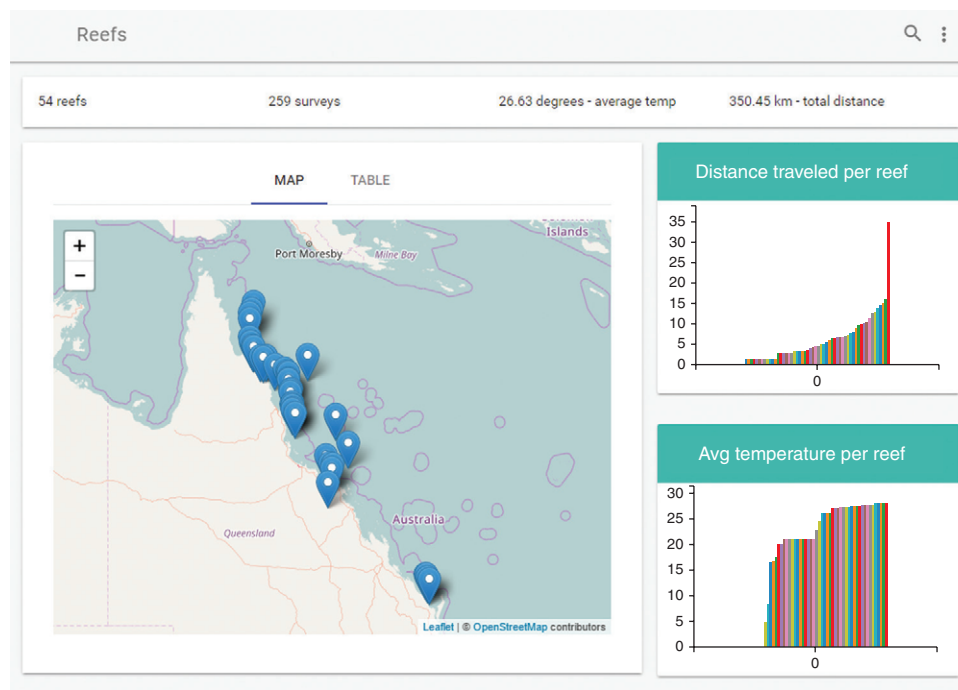


Figure 1.16 The reef data dashboard

With a plan and direction, you might engage in test-driven development or another form of automated testing (more on that in chapter 14).

The dashboard also has a table of reefs where you can drill down for a closer look (figure 1.17). To make the data display efficiently in the dashboard, I've prebaked various data analysis into the database.

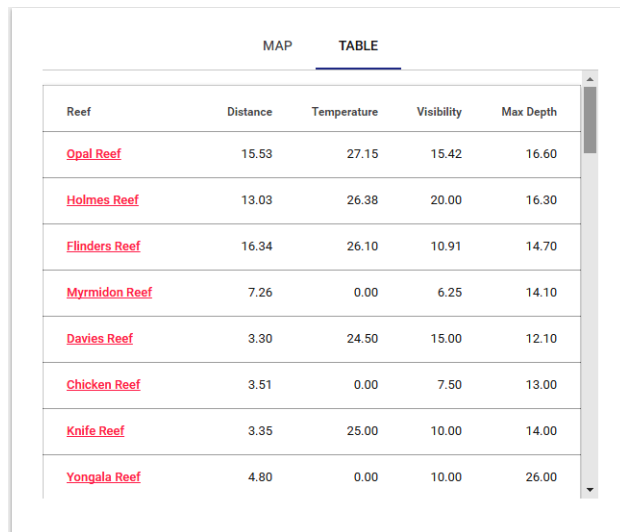
To get your code into production, you'll most likely need a form of build or deployment script, maybe both. The build script will do such things as static error checking, concatenation, minification, and packaging your code for deployment. Your deployment script takes your code and copies it to the environment where it will run. You typically need a deployment script when you're deploying a server or microservice. To host your server in the cloud, you may also need a provisioning script. This is a script that creates the environment in which the code will run. It might create a VM from an image and then install dependencies—for example, Node.js and MongoDB.

With your code moved to the production environment, you have a whole new set of issues to deal with:

- What happens when you get data updates that don't fit your initial assumptions?
- What happens when your code crashes?
- How do you know if your code is having problems?
- What happens when your system is overloaded?

You'll explore these issues and how to approach them in chapter 14.

Welcome to the world of data wrangling. You now have an understanding of what a data-wrangling project might look like, and you'll spend the rest of the book exploring the various phases of the process, but before that, you might need help getting started with Node.js, so that's what we'll cover in chapter 2.



MAP		TABLE		
Reef	Distance	Temperature	Visibility	Max Depth
<a href="#">Opal Reef</a>	15.53	27.15	15.42	16.60
<a href="#">Holmes Reef</a>	13.03	26.38	20.00	16.30
<a href="#">Flinders Reef</a>	16.34	26.10	10.91	14.70
<a href="#">Myrmidon Reef</a>	7.26	0.00	6.25	14.10
<a href="#">Davies Reef</a>	3.30	24.50	15.00	12.10
<a href="#">Chicken Reef</a>	3.51	0.00	7.50	13.00
<a href="#">Knife Reef</a>	3.35	25.00	10.00	14.00
<a href="#">Yongala Reef</a>	4.80	0.00	10.00	26.00

**Figure 1.17** Table of reefs in the dashboard

## Summary

- Data wrangling is the entire process of working with data from acquisition through processing and analysis, then finally to reporting and visualization.
- Data analysis is a part of data wrangling, and it *can* be done in JavaScript.
- JavaScript is already a capable language and is improving with each new iteration of the standard.
- As with any coding, data wrangling can be approached in a range of ways. It has a spectrum from ad hoc throw-away coding to disciplined high-quality coding. Where you fit on this spectrum depends on the time you have and the intended longevity of the code.
- Exploratory coding is important for prototyping code and understanding data.
- Data wrangling has a number of phases: acquisition, cleanup, transformation, then analysis, reporting, and visualization.
- The phases are rarely cleanly separated; they're often interspersed and tangled up with each other.
- You should always start with planning.
- It's important to check assumptions about the data.
- Moving code to production involves many new issues.

# Data Wrangling with JavaScript

Ashley Davis



**W**hy 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  
[mannning.com/books/data-wrangling-with-javascript](http://mannning.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]