





Reading Sample

In these chapters, you'll explore a selection of key Node.js topics. You'll take your first steps with the Node.js fundamentals, and then explore the Express web application framework. Finally, you'll walk through creating command-line applications with Node.js.

-  **"Basic Principles"**
-  **"Express"**
-  **"Node on the Command Line"**
-  **Contents**
-  **Index**
-  **The Author**

Sebastian Springer

Node.js: The Comprehensive Guide

834 pages, 2022, \$49.95

ISBN 978-1-4932-2292-6

 www.rheinwerk-computing.com/5556

Chapter 1

Basic Principles

All beginnings are difficult.
—Ovid

Bringing more dynamics into web pages was the original idea behind JavaScript. The scripting language was intended to compensate for the weaknesses of HTML when it came to responding to user input. The history of JavaScript dates back to 1995 when it was developed under the code name Mocha by Brendan Eich, a developer at Netscape. One of the most remarkable facts about JavaScript is that the first prototype of this successful and globally used language was developed in just 10 days. Still in the year of its creation, Mocha was renamed to LiveScript and finally to JavaScript in a cooperation between Netscape and Sun. This was mainly for marketing purposes, as at that time it was assumed that Java would become the leading language in client-side web development.

Node.js

ES2015

Support

Node.js

language's compat-table applied only to Node.js

Learn more

Nightly!

11.0.0

10.1.0

9.11.1

8.9.4

8.6.0

8.2.1

7.10.1

7.5.0

6.14.2

6.4.0

5.12.0

4.9.1

0.12.18

0.10.48

100% complete

99% complete

99% complete

99% complete

99% complete

99% complete

99% complete

99% complete

99% complete

99% complete

99% complete

97% complete

31% complete

11% complete

optimisation

proper tail calls (tail call optimisation)

direct recursion

mutual recursion

syntax

default function parameters

basic functionality

explicit undefined defers to the default

defaults can refer to previous params

arguments object interaction

temporal dead zone

separate scope

new Function() support

rest parameters

basic functionality

function 'length' property

arguments object interaction

can't be used in setters

new Function() support

spread (...) operator

with arrays, in function calls

with arrays, in array literals

with sparse arrays, in function calls

with sparse arrays, in array literals

with strings, in function calls

with strings, in array literals

with astral plane strings, in function calls

with astral plane strings, in array literals

with generator instances, in calls

with generator instances, in arrays

with generic iterables, in calls

with generic iterables, in arrays

with instances of iterables, in calls

with instances of iterables, in arrays

spreading non-iterables is a runtime error

Figure 1.1 Support for JavaScript Features in Node.js (<http://node.green>)

Convinced by the success of JavaScript, Microsoft also integrated a scripting language into Internet Explorer 3 in 1996. This was the birth of JScript, which was mostly compatible with JavaScript, but with additional features added.

Today, the mutual vying of the two companies is known as the “browser wars.” The development ensured that the two JavaScript engines steadily improved in both feature set and performance, which is the primary reason for JavaScript’s success today.

In 1997, the first draft of the language standard was created at Ecma International. The entire language core of the script language is recorded under the cryptic designation ECMA-262 or ISO/IEC 16262. The current standard can be found at www.ecma-international.org/publications/standards/Ecma-262.htm. Due to this standardization, vendor-independent JavaScript is also referred to as ECMAScript. Until a few years ago, the ECMAScript standard was versioned in integers starting at 1. Since version 6, the versions are also provided with year numbers. ECMAScript in version 8 is therefore referred to as ECMAScript 2017. As a rule, you can assume that the manufacturers support the older versions of the standard quite well. You must either enable newer features by configuration flags in the browser or simulate them via polyfills (that is, recreating the features in JavaScript). A good overview of the currently supported features is provided by kangax’s compatibility table, which can be found at <http://kangax.github.io/compat-table/es6/>. A version adapted for Node.js can be reached at <http://node.green/>.

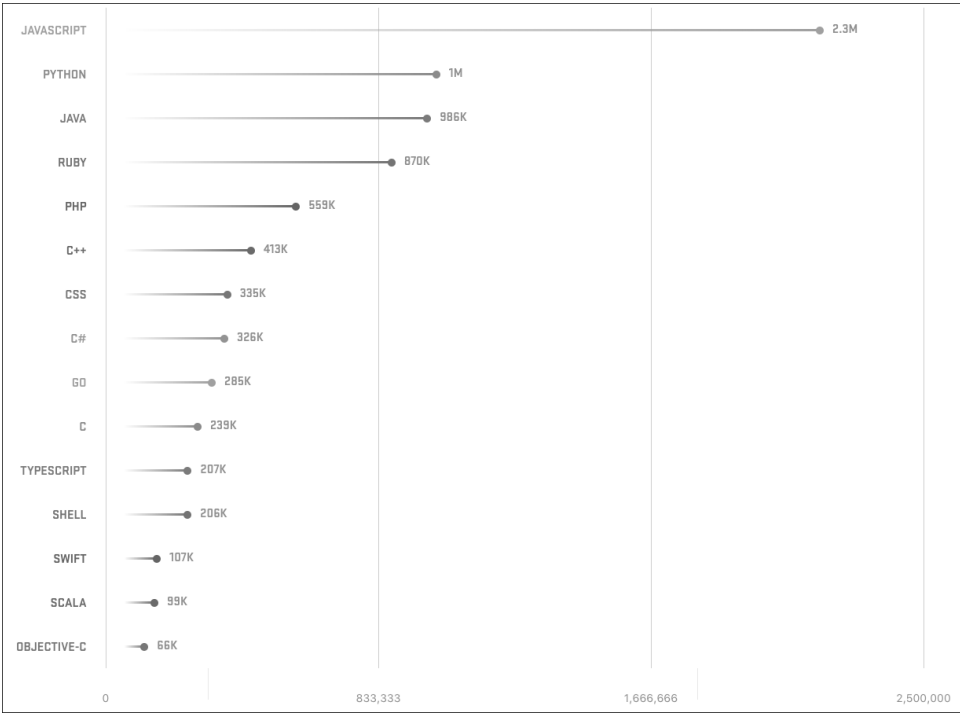


Figure 1.2 Top Languages in GitHub Based on Pull Requests (octoverse.github.com)

JavaScript is lightweight, is relatively easy to learn, and has a huge ecosystem of frameworks and libraries. For these reasons, JavaScript is one of the most successful programming languages in the world. This success can be backed up by numbers: Since 2008, JavaScript has been in the top two spots in GitHub’s language trends. In 2021, JavaScript was passed by Python in the number one spot and is now in second place in language trends.

Node.js is based on this successful scripting language and has had a meteoric rise itself. This chapter will serve as an introduction to the world of Node.js, showing you how the platform is built, and where and how you can use Node.js.

1.1 The Story of Node.js

To help you better understand what Node.js is and how some of the development decisions came about, let’s explore the history of the platform.

1.1.1 Origins

Node.js was originally developed by Ryan Dahl, a PhD student in mathematics who thought better of it, abandoned his efforts, and instead preferred to travel to South America with a one-way ticket and very little money in his pocket. There, he kept his head above water by teaching English. During this time, he got in touch with PHP as well as Ruby and discovered his affection for web development. The problem with working with the Ruby framework, called Rails, was that it couldn’t deal with concurrent requests without any workaround. The applications were too slow and utilized the CPU entirely. Dahl found a solution to his problems with Mongrel, a web server for applications based on Ruby.

Unlike traditional web servers, Mongrel responds to user requests and generates responses dynamically, where otherwise only static HTML pages are delivered.

The task that actually led to the creation of Node.js is quite trivial from today’s point of view. In 2005, Dahl was looking for an elegant way to implement a progress bar for file uploads. However, the technologies available at the time only allowed unsatisfactory solutions. Regarding file transfers, HTTP was used for relatively small files, and File Transfer Protocol (FTP) was used for larger files. The status of the upload was queried using long polling, which is a technique where the client sends long-lived requests to the server, and the server uses the open channel for replies. Dahl’s first attempt to implement a progress bar took place in Mongrel. After sending the file to the server, it checked the status of the upload using a large number of Asynchronous JavaScript and XML (AJAX) requests and displayed it graphically in a progress bar. However, the downside of this implementation was Ruby’s single-threaded approach and the large number of requests that were required.

Another promising approach involved an implementation in C. Here, Dahl's options weren't limited to one thread. However, C as a programming language for the web has a decisive disadvantage: only a small number of developers are enthusiastic about this field of application. Dahl was also confronted with this problem and discarded this approach after a short time.

The search for a suitable programming language to solve his problem continued and led him to functional programming languages such as Haskell. Haskell's approach is built on nonblocking input/output (I/O), which means that all read and write operations are asynchronous and don't block the execution of a program. This allows the language to remain single-threaded at its core and doesn't introduce the problems that arise from parallel programming. Among other things, no resources have to be synchronized, and no problems are caused by the runtime of parallel threads. However, Dahl still wasn't fully satisfied with this solution and was looking for other options.

1.1.2 Birth of Node.js

Dahl then found the solution he was finally satisfied with—JavaScript. He realized that this scripting language could meet all his requirements. JavaScript had already been established on the web for years, so there were powerful engines and a large number of developers. In January 2009, he began working on his implementation for server-side JavaScript, which can be regarded as the birth of Node.js. Another reason for implementing the solution in JavaScript, according to Dahl, was the fact that the developers of JavaScript didn't envision this area of use. At that time, no native web server existed in JavaScript, it couldn't handle files in a file system, and there was no implementation of sockets to communicate with other applications or systems. All these points spoke in favor of JavaScript as the basis for a platform for interactive web applications because no determinations had yet been made in this area, and, consequently, no mistakes had yet been made either. The architecture of JavaScript also argued for such an implementation. The approach of top-level functions (i.e., functions that aren't linked to any object, are freely available, and can be assigned to variables) offers a high degree of flexibility in development and enables functional approaches to solutions.

Thus, Dahl selected other libraries in addition to the JavaScript engine, which is responsible for interpreting the JavaScript source code, and put them together in one platform.

In September 2009, Isaac Schlueter started working on a package manager for Node.js, the *Node Package Manager* (*npm*).

1.1.3 Breakthrough of Node.js

After Dahl integrated all the components and created the first executable examples on the new Node.js platform, he needed a way to introduce Node.js to the public. This also

became necessary because his financial resources shrank considerably due to the development of Node.js, and he would have had to stop working on Node.js if he didn't find any sponsors. He chose the JavaScript conference JSConf EU in November 2009 in Berlin as his presentation platform. Dahl put all his eggs in one basket. If the presentation was a success and he found sponsors to support his work on Node.js, he could continue his involvement; if not, almost a year's work would have been in vain. In a rousing talk, he introduced Node.js to the audience and showed how to create a fully functional web server with just a few lines of JavaScript code. As another example, he introduced an implementation of an Internet Relay Chat (IRC) chat server. The source code for this demonstration comprised about 400 lines. Using this example, he demonstrated the architecture and thus the strengths of Node.js while making it tangible for the audience. The recording of this presentation can be found at www.youtube.com/watch?v=EeYvFl7-li9E. The presentation didn't miss its mark and led to Joyent stepping in as a sponsor for Node.js. Joyent is a San Francisco-based software and services provider offering hosting solutions and cloud infrastructure. With its commitment, Joyent included the open-source software Node.js in its product portfolio and made Node.js available to its customers as part of its hosting offerings. Dahl was hired by Joyent and became a full-time maintainer for Node.js from that point on.

1.1.4 Node.js Conquers Windows

The developers made a significant step toward the spread of Node.js by introducing native support for Windows in version 0.6 in November 2011. Up to that point, Node.js could only be installed awkwardly on Windows via Cygwin.

Since version 0.6.3 in November 2011, npm has been an integral part of the Node.js packages and is thus automatically delivered when Node.js is installed.

Surprisingly, at the start of 2012, Dahl announced that he would finally retire from active development after three years of working on Node.js. He handed over the reins of development to Schlueter. The latter, like Dahl, was an employee at Joyent and actively involved in the development of the Node.js core. The change unsettled the community, as it wasn't clear whether the platform would continue to develop without Dahl. A signal that the Node.js community considered as being strong enough for solid further development came with the release of version 0.8 in June 2012, which was primarily intended to significantly improve the performance and stability of Node.js.

With version 0.10 in March 2013, one of the central interfaces of Node.js changed: the Stream application programming interface (API). With this change, it became possible to actively pull data from a stream. Because the previous API was already widely used, both interfaces continued to be supported.

1.1.5 io.js: The Fork of Node.js

In January 2014, there was another change in the project management of Node.js. Schluter, who left Node.js maintenance in favor of his own company (called npmjs), the host of the npm repository, was succeeded by TJ Fontaine. Under his direction, version 0.12 was released in February 2014. A common criticism of Node.js at the time was that the framework had still not reached the supposedly stable version 1.0, which prevented numerous companies from using Node.js for critical applications.

Many developers were unhappy with Joyent, which had provided maintainers for Node.js since Dahl, and so the community fractured in December 2014. The result was io.js, a fork of Node.js that was developed separately from the original platform. As a result, the independent Node.js Foundation was founded in February 2015, which was responsible for the further development of io.js. At the same time, version 0.12 of the Node.js project was released.

1.1.6 Node.js Reunited

In June 2015, the two projects io.js and Node.js were merged into the Node.js Foundation. With version 4 of the project, the merger was completed. Further development of the Node.js platform is now coordinated by a committee within the Node.js Foundation rather than by individuals. As a result, we see more frequent releases and a stable version with long-term support (LTS).

1.1.7 Deno: A New Star in the JavaScript Sky

Since the merger of io.js and Node.js, things have become quieter around Node.js. The regular releases, the stability, and also the integration of new features, such as worker threads, HTTP/2 or performance hooks, keep up the good mood within the community. And just when things were starting to get almost too quiet around Node.js, an old acquaintance, Dahl, took the stage again in 2018 to introduce a new JavaScript platform called Deno during his talk, “10 Things I Regret about Node.js.”

The idea behind Deno is to create a better Node.js, untethered from the backwards compatibility constraints that prevent revolutionary leaps in development. For example, Deno is based on TypeScript by default and adds a fundamentally different module system. Deno’s core is also quite different from Node.js, as it’s written almost entirely in Rust.

Nevertheless, there are also some common features. For example, Deno is based on the tried and tested V8 engine, which also forms the heart of Node.js. And you don’t have to do without the huge number of npm packages either. For this purpose, Deno provides a compatibility layer. You can read more about Deno in Chapter 28.

1.1.8 OpenJS Foundation

In 2015, the Node.js Foundation was established to coordinate the development of the platform. The foundation was a subordinate project of the Linux Foundation. In 2019, the JS Foundation and the Node.js Foundation then merged to form the OpenJS Foundation. In addition to Node.js, it includes a number of other popular projects such as webpack, ESLint, and Electron.

1.2 Organization of Node.js

The community behind Node.js has learned its lessons from the past. For this reason, there are no longer individuals at the helm of Node.js, but a committee of several people who steer the development of the platform.

1.2.1 Technical Steering Committee

The technical steering committee (TSC) is responsible for further developing the platform. The number of members of the TSC isn’t limited, but 6 to 12 members are targeted, usually selected from the contributors to the platform. The tasks of the TSC are as follows:

- Setting the technical direction of Node.js
- Performing project and process control
- Defining the contribution policy
- Managing the GitHub repository
- Establishing the conduct guidelines
- Managing the list of collaborators

The TSC holds weekly meetings via Google Hangouts to coordinate and discuss current issues. Many of these meetings are published via the Node.js YouTube channel (www.youtube.com/c/nodejs+foundation).

1.2.2 Collaborators

Node.js is an open-source project developed in a GitHub repository. As with all larger projects of this type, a group of people, called collaborators, have write access to this repository. In addition to accessing the repository, a collaborator can access the continuous integration jobs. Typical tasks of a collaborator include supporting users and new collaborators, improving Node.js source code and documentation, reviewing pull requests and issues (with appropriate commenting), participating in working groups, and merging pull requests.

Collaborators are designated by the TSC. Usually the role of a collaborator is preceded by a significant contribution to the project via a pull request.

1.2.3 Community Committee

As the name implies, the Community Committee (CommComm) takes care of the Node.js community with a special focus on education and culture. The CommComm coordinates in regular meetings, which are recorded in a separate GitHub repository (<https://github.com/nodejs/community-committee>). The CommComm exists to give the community a voice and thus counterbalance the commercial interests of corporations.

1.2.4 Work Groups

The TSC establishes various work groups to have specific topics addressed separately by experts. Examples of such work groups include the following:

- **Release**
This work group manages the release process of the Node.js platform, defining the content of the releases and taking care of LTS.
- **Streams**
The streams work group is working to improve the platform’s Stream API.
- **Docker**
This work group manages the official Docker images of the Node.js platform and ensures that they are kept up to date.

1.2.5 OpenJS Foundation

The OpenJS Foundation forms the umbrella for Node.js development. Its role is similar to that of the Linux Foundation for the development of the Linux operating system. The OpenJS Foundation was founded as an independent body for further developing Node.js. Its list of founding members includes companies such as IBM, Intel, Joyent, and Microsoft. The OpenJS Foundation is funded by donations and contributions from companies and individual members.

1.3 Versioning of Node.js

One of the biggest points of criticism concerning Node.js before the fork of io.js was that its development was very slow. Regular and predictable releases are an important selection criterion, especially in enterprise usage. For this reason, after merging Node.js and io.js, the developers of Node.js agreed on a transparent release schedule with regular

releases and an LTS version that is provided with updates over a longer period of time. The release schedule provides for one major release per half year.

Table 1.1 shows the release schedule of Node.js.

Release	Status	Initial Release	Active LTS Start	Maintenance LTS Start	End of Line
https://nodejs.org/download/release/latest-v14.x/	Maintenance LTS	4/21/2020	10/27/2020	10/19/2021	4/30/2023
https://nodejs.org/download/release/latest-v16.x/	Active LTS	4/20/2021	10/26/2021	10/18/2022	4/30/2024
https://nodejs.org/download/release/latest-v17.x/	Current	10/19/2021		4/1/2022	6/1/2022
https://nodejs.org/download/release/latest-v18.x/	Pending	4/19/2022	10/25/2022	10/18/2023	4/30/2025
v19	Pending	10/18/2022		4/1/2023	6/1/2023
v20	Pending	4/18/2023	10/24/2023	10/22/2024	4/30/2026

Table 1.1 Node.js Release Schedule

As you can see from the release schedule, versions with an even version number are LTS releases, while odd ones are releases with a shortened support period.

1.3.1 Long-Term Support Releases

A Node.js version with an even version number is transitioned to an LTS release as soon as the next odd version is released. The LTS release is then actively maintained over a period of 12 months. During this time, the version receives the following:

- Bug fixes
- Security updates
- npm minor updates
- Documentation updates
- Performance improvements that don’t compromise existing applications
- Changes to the source code that simplify the integration of future improvements

After this phase, the version enters a 12-month maintenance phase during which the version will continue to receive security updates. In this case, however, only critical bugs and security gaps are fixed. In total, the developers of the Node.js platform support an LTS release over a period of 30 months.

1.4 Benefits of Node.js

The development history of Node.js shows one thing very clearly: it's directly connected to the internet. With JavaScript as its base, you have the ability to achieve visible results very quickly with applications implemented in Node.js. The platform itself is very lightweight and can be installed on almost any system. As is common for a scripting language, Node.js applications also omit a heavyweight development process, so you can check the results directly. In addition to the fast initial implementation, you can also react very flexibly to changing requirements during the development of web applications. Because the core of JavaScript is standardized by ECMAScript, the language represents a reliable basis with which even more extensive applications can be implemented. The available language features are well documented extensively both online and in reference books. In addition, many developers are proficient in JavaScript and able to implement even larger applications using this language. Because Node.js uses the same JavaScript engine as Google Chrome—the V8 engine—all language features are also available here, and developers who are proficient in JavaScript can familiarize themselves with the new platform relatively quickly.

JavaScript's long history of development has produced a number of high-performance engines. One reason for this development is that the various browser manufacturers were always developing their own implementations of JavaScript engines, so there was healthy competition in the market when it came to running JavaScript in the browser. On one hand, this competition led to the fact that JavaScript is now interpreted very quickly, and, on the other hand, it led to manufacturers agreeing on certain standards. Node.js as a platform for server-side JavaScript was designed as an open-source project since the beginning of its development. For this reason, an active community quickly developed around the core of the platform and deals mainly with the use of Node.js in practice, but also with the further development and stabilization of the platform. Resources on Node.js range from tutorials to help you get started to articles on advanced topics such as quality assurance, debugging, or scaling. The biggest advantage of an open-source project such as Node.js is that the information is available to you free of charge, and questions and problems can be solved quite quickly and competently via a wide variety of communication channels or the community itself.

1.5 Areas of Use for Node.js

From a simple command-line tool to an application server for web applications running on a cluster with numerous nodes, Node.js can be used anywhere. The use of a technology strongly depends on the problem to be solved, personal preferences, and the developers' level of knowledge. For this reason, not only should you know the key features of Node.js, but you should also have a feel for working with the platform. You can only fulfill the second point if you either have the opportunity to join an existing

Node.js project or gain the experience in the best case with smaller projects that you implement.

But let's now turn to the most important framework data:

■ Pure JavaScript

When working with Node.js, you don't have to learn a new language dialect because you can fall back on the JavaScript language core. Standardized and well-documented interfaces are available for accessing system resources. However, as an alternative to JavaScript, you can also write your Node.js application in TypeScript, translate the source code to JavaScript, and run it with Node.js. You'll find more information about this topic in Chapter 13.

■ Optimized engine

Node.js is based on Google's V8 JavaScript engine. Here, you benefit above all from the constant further development of the engine, where the latest language features are supported already after a very short time.

■ Nonblocking I/O

All operations that don't take place directly in Node.js don't block the execution of your application. The principle of Node.js is that everything the platform doesn't have to do directly is outsourced to the operating system, other applications, or other systems. This gives the application the ability to respond to additional requests or to process tasks in parallel. Once the processing of a task is complete, the Node.js process receives feedback and can process the information further.

■ Single-threaded

A typical Node.js application runs in a single process. For a long time, there hasn't been any multithreading, and concurrency was initially only provided for in the form of the nonblocking I/O already described. Thus, all the code you write yourself potentially blocks your application. For this reason, you should pay attention to resource-saving development. If it still becomes necessary to process tasks in parallel, Node.js offers you solutions for this in the form of the `child_process` module, which enables you to create your own child processes.

To develop your application in the best possible way, you should have at least a rough overview of the components and how they work. The most important of these components is the V8 engine.

1.6 The Core: V8 Engine

For you, as a developer, to assess whether a technology can be used in a project, you should be sufficiently familiar with the characteristics of that technology. The sections that follow now dive deep into the internal details of Node.js to show you the components that make up the platform and how you can use them to the advantage of an application.

The central and thus most important component of the Node.js platform is the V8 JavaScript engine developed by Google (for more information, visit the V8 Project page at <https://code.google.com/p/v8/>). The JavaScript engine is responsible for interpreting and executing the JavaScript source code. There isn't just one engine for JavaScript; instead, the different browser manufacturers use their own implementations. One of the problems with JavaScript is that each engine behaves slightly differently. Standardization to ECMAScript attempts to find a reliable common denominator so that you, as a JavaScript application developer, have less uncertainty to worry about. The competition among JavaScript engines resulted in a number of optimized engines, all with the goal of interpreting JavaScript code as quickly as possible. Over time, several engines have established themselves on the market: Mozilla's JaegerMonkey, Apple's Nitro, and Google's V8 engine, among others. Microsoft meanwhile uses the same technical basis as Chrome for its Edge browser, so it also uses the V8 engine.

Node.js uses Google's V8 engine. This engine has been developed by Google since 2006, mainly in Denmark, in collaboration with Aarhus University. The engine's primary area of use is Google's Chrome browser, where it's responsible for interpreting and executing JavaScript code. The goal of developing a new JavaScript engine was to significantly improve the performance of interpreting JavaScript. The engine now fully implements the ECMAScript standard ECMA-262 in the fifth version and large parts of the sixth version. The V8 engine itself is written in C++, runs on various platforms, and is available under the Berkeley Source Distribution (BSD) license as open-source software for any developer to use and improve. For example, you can integrate the engine into any C++ application.

As usual in JavaScript, the source code isn't compiled before execution; instead, the files containing the source code are read directly when the application is launched. Launching the application starts a new Node.js process. This is where the first optimization by the V8 engine takes place. The source code isn't directly interpreted, but is first translated into machine code, which is then executed. This technology is referred to as just-in-time (JIT) compilation and is used to increase the execution speed of the JavaScript application. The actual application is then executed on the basis of the compiled machine code. The V8 engine makes further optimizations in addition to JIT compilation. Among other things, these include improved garbage collection and an improvement in the context of accessing object properties. For all the optimizations that the JavaScript engine makes, you should keep in mind that the source code is read at process startup, so the changes to the files have no effect on the running application. For your changes to take effect, you must exit and restart your application so that the customized source code files are read again.

1.6.1 Memory Model

The goal of developing the V8 engine was to achieve the highest possible speed in the execution of JavaScript source code. For this reason, the memory model has also been

optimized. Tagged pointers, which are references in memory that are marked as such in a special way, are used in the V8 engine. All objects are 4-byte-aligned, which means that 2 bits are available to identify pointers. A pointer always ends on 01 in the memory model of the V8 engine, whereas a normal integer value ends on 0. This measure allows integer values to be distinguished very quickly from memory references, which provides an extremely significant performance advantage. The object representations of the V8 engine in memory each consist of three data words. The first data word consists of a reference to the hidden class of the object, which you'll learn more about in later sections. The second data word is a pointer to the attributes, that is, the properties of the object. Finally, the third data word refers to the elements of the object. These are the properties with a numeric key. This structure supports the JavaScript engine in its work and is optimized in such a way that elements in the memory can be accessed very fast so that little wait time arises from searching objects.

1.6.2 Accessing Properties

As you probably know, JavaScript doesn't know classes; the object model of JavaScript is based on prototypes. In class-based languages such as Java or PHP, classes represent the blueprint of objects. These classes can't be changed at runtime. Prototypes in JavaScript, on the other hand, are dynamic, which means that properties and methods can be added and removed at runtime. As with all other languages that implement the object-oriented programming paradigm, objects are represented by their properties and methods, where properties represent the state of an object, and methods are used to interact with the object. In an application, you usually access the properties of the various objects very frequently. In addition, methods in JavaScript are also properties of objects that are stored with a function. In JavaScript, you work almost exclusively with properties and methods, so access to them must be very fast.

Prototypes in JavaScript

JavaScript differs from languages such as C, Java, or PHP in that it doesn't take a class-based approach but instead is based on prototypes, such as the Self language. In JavaScript, every object normally has a `prototype` property and thus a prototype. In JavaScript, as in other languages, you can create objects. For this purpose, however, you don't use classes in conjunction with the `new` operator. Instead, you can create new objects in several different ways. Among other things, you can use constructor functions or the `Object.create` method. These methods have in common that you create an object and assign the prototype. The prototype is an object from which another object inherits its properties. Another feature of prototypes is that they can be modified at application runtime, allowing you to add new properties and methods. By using prototypes, you can build an inheritance hierarchy in JavaScript.

Normally, accessing properties in a JavaScript engine is done through a directory in the memory. So, if you access a property, this directory is searched for the memory section of the respective property, and then the value can be accessed. Now imagine a large application that maps its business logic in JavaScript on the client side, and in which a large number of objects are held in parallel in the memory, constantly communicating with each other. This method of accessing properties would quickly turn into a problem. The developers of the V8 engine have recognized this vulnerability and developed a solution for it—the hidden classes. The real problem with JavaScript is that the structure of objects is only known at runtime and not already during the compilation process because such a process doesn't exist with JavaScript. This is further complicated by the fact that there isn't just one prototype in the structure of objects, but they can rather exist in a chain. In classical languages, the object structure doesn't change at application runtime; the properties of objects are always located in the same place, which significantly speeds up accessing them.

A hidden class is nothing more than a description in which the individual properties of an object can be found in the memory. For this purpose, a hidden class is assigned to each object. This contains the offset to the memory section within the object where the respective property is stored. As soon as you access a property of an object, a hidden class is created for that property and reused for each subsequent access. So for an object, there is potentially a separate hidden class for each property.

In Listing 1.1, you can see an example that illustrates how hidden classes work.

```
class Person {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}
const johnDoe = new Person("John", "Doe");
```

Listing 1.1 Accessing Properties in a Class

In the example, you create a new constructor function for the group of `person` objects. This constructor has two parameters—the first name and the last name of the person. These two values are to be stored in the `firstname` and `lastname` properties of the object, respectively. When a new object is created with this constructor using the `new` operator, an initial hidden class, class 0, is created first. This doesn't yet contain any pointers to properties. If the first assignment is made, that is, the first name is set, a new hidden class, class 1, is created based on class 0. This now contains a reference to the memory section of the `firstname` property, relative to the beginning of the object's namespace. In addition, a class transition is added to class 0, which states that class 1 should be used instead of class 0 if the `firstname` property is added. The same process takes place when the second assignment is performed for the last name. Another hidden class, class 2, is

created based on class 1, which then contains the offset for both the `firstname` and `lastname` properties and inserts a transition indicating that class 2 should be used when the `lastname` property is used. If properties are added away from the constructor, and this is done in a different order, new hidden classes are created in each case. Figure 1.3 clarifies this process.

When the properties of an object are accessed for the first time, the use of hidden classes doesn't yet result in a speed advantage. However, all subsequent accesses to the property of the object then happen many times faster, because the engine can directly use the hidden class of the object and this contains the reference to the memory section of the property.

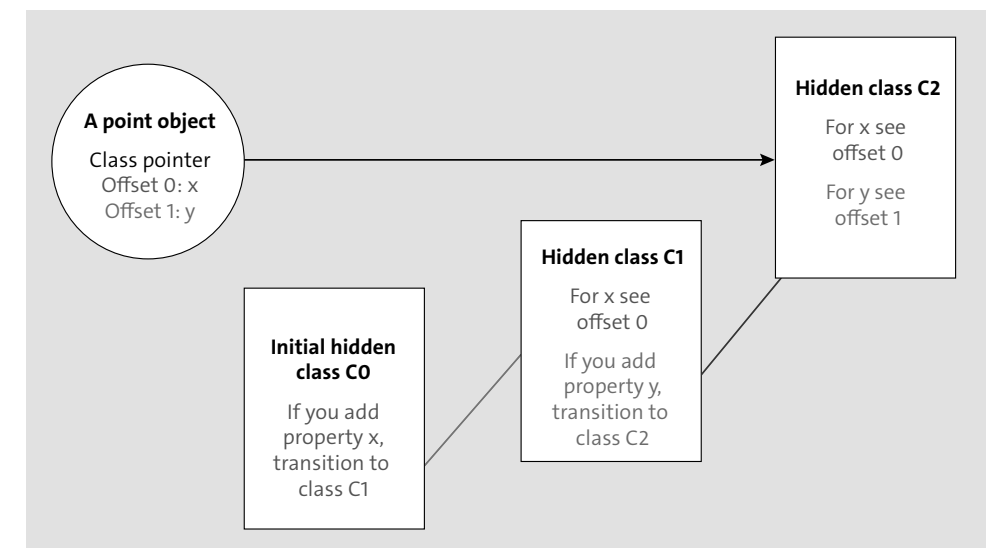


Figure 1.3 Hidden Classes in the V8 Engine (<https://github.com/v8/v8/wiki/Design%20Elements#fast-property-access>)

1.6.3 Machine Code Generation

As you already know, the V8 engine doesn't directly interpret the JavaScript application source code, but performs a JIT compilation into native machine code to increase execution speed. No optimizations are made to the source code during this compilation. The source code written by the developer is thus converted one to one. In addition to this JIT compiler, the V8 engine has another compiler that is capable of optimizing the machine code. To decide which code fragments to optimize, the engine maintains internal statistics about the number of function calls and how long each function is executed. Based on this data, the decision is made regarding whether the machine code of a function requires optimizing.

Now you're probably wondering why the entire source code of the application isn't compiled with the second, much better compiler. There is a very simple reason for this:

a compiler that doesn't perform optimizations is much faster. Because the source code is compiled JIT, this process is very time critical because any wait times caused by a compilation process that takes too long can have a direct impact on the user. Therefore, only code sections that justify this additional effort are optimized. This machine code optimization has a particularly positive effect on larger and longer-running applications and on those in which functions are called more often than just once.

Another optimization the V8 engine performs is related to the hidden classes and internal caching already described earlier. After the application is launched and the machine code is generated, the V8 engine searches for the associated hidden class each time a property is accessed. As a further optimization, the engine assumes that the objects used at this point will have the same hidden class in the future, so it modifies the machine code accordingly. The next time the code section is traversed, the property can be accessed directly with no need to search for the associated hidden class first. If the object used doesn't have the same hidden class, the engine detects this, removes the previously generated machine code, and replaces it with the corrected version. There is a critical problem with this approach: Imagine you have a code section where two different objects with different hidden classes are always used in alternation. Then the optimization with the prediction of the hidden class would never take effect at the next execution. In this case, various code fragments are used, which can't be used to find the memory section of a property as quickly as with just one hidden class, but the code in this case is many times faster than without the optimization because it's usually possible to select from a very small set of hidden classes. The generation of machine code and the hidden classes in combination with the caching mechanisms creates possibilities that are familiar from class-based languages.

1.6.4 Garbage Collection

The optimizations described so far mainly affect the speed of an application. Another very important feature is the garbage collector of the V8 engine. *Garbage collection* refers to the process of clearing up the application's memory area in the main memory. Elements that are no longer used are removed from memory so that the space freed up becomes available to the application again.

If you're wondering why you need a garbage collector in JavaScript, the answer is quite simple: Originally, JavaScript was intended for small tasks on web pages. These web pages, and thus the JavaScript on this page, had a fairly short lifetime until the page was reloaded, completely emptying the memory containing the JavaScript objects. The more JavaScript is executed on a page and the more complex the tasks to be performed become, the greater the risk that memory will be filled with objects that are no longer needed. If you now assume you have an application in Node.js that has to run for several days, weeks, or even months without restarting the process, the problem becomes clear. The V8 engine's garbage collector comprises a number of features that allow it to

perform its tasks very quickly and efficiently. Basically, when the garbage collector is running, the engine stops the execution of the application completely and resumes it as soon as the run is finished. These application pauses are in the single-digit millisecond range so that the user normally doesn't feel any negative effects due to the garbage collector. To keep the interruption by the garbage collector as short as possible, the complete memory isn't cleaned up, but only parts of it. In addition, the V8 engine knows at all times where in the memory which objects and pointers are located.

The V8 engine divides the available memory into two areas—one area for storing objects and another area to keep the information about the hidden classes and the executable machine code. The process of garbage collection is relatively simple. When an application is executed, objects and pointers are created in the short-lived area of the V8 engine's memory. If this memory area is full, it's cleaned up. Objects that are no longer used are deleted, and objects that are still needed are moved to the long-lived area. During this shift, the object itself is shifted, and the pointers to the object's memory location are corrected. The partitioning of memory areas makes different types of garbage collection necessary.

The fastest variant is represented by the scavenge collector, which is very fast and efficient and deals only with the short-lived area. Two different garbage collection algorithms exist for the long-lived memory section, both based on mark-and-sweep. The entire memory is searched, and elements that are no longer needed are marked and later deleted. The real problem with this algorithm is that it creates gaps in the memory, which causes problems over a longer runtime of an application. For this reason, a second algorithm exists that also searches the elements of the memory for those that are no longer needed, marks them, and deletes them.

The most important difference between the two is that the second algorithm defragments the memory; that is, it rearranges the remaining objects in the memory so that afterwards, the memory has as few gaps as possible. This defragmentation can only happen because V8 knows all objects and pointers. For all its benefits, the garbage collection process also has a drawback: it takes time. The fastest the scavenge collection can run is about 2 ms. This is followed by the mark-and-sweep process without optimizations at 50 ms and finally the mark-and-sweep with defragmentation with an average of 100 ms.

In the following sections, you'll learn more about the other elements used in the Node.js platform besides the V8 engine.

1.7 Libraries around the Engine

The JavaScript engine alone doesn't make a platform yet. For Node.js to handle all requirements such as event handling, I/O, or support functions such as Domain Name System (DNS) resolution or encryption, additional functionality is required. This is

implemented with the help of additional libraries. For many tasks that a platform such as Node.js has to deal with, ready-made and established solutions already exist. For this reason, Dahl decided to build the Node.js platform on top of a set of external libraries and fill in the gaps he felt weren't adequately covered by any existing solution with his own implementations. The advantage of this strategy is that you don't have to reinvent the solutions for standard problems; you can fall back on tried and tested libraries.

A prominent example that is also built on this strategy is the Unix operating system. In this context, developers should stick to the following principle: focus only on the actual problem, solve it as well as possible, and use existing libraries for everything else. Most command-line programs in the Unix area implement this philosophy. Once a solution has established itself, it can be used in other applications for similar problems. This in turn has the advantage that improvements in the algorithm only have to be made at one central point. The same applies to bug fixes. If an error occurs in DNS resolution, it's fixed once, and the solution works in all places where the library is used. But that also leads to the flip side of the coin: the libraries on which the platform is built must exist. Node.js solves this problem in that it's built on only a small set of libraries that must be provided by the operating system. But these dependencies rather consist of basic functions such as the GNU Compiler Collection (GCC) runtime library or the standard C library. The remaining dependencies, such as `zlib` or `http_parser`, are included in the source code.

1.7.1 Event Loop

Client-side JavaScript contains many elements of an event-driven architecture. Most user interactions cause events that are responded to with appropriate function calls. By using various features such as first-class functions and anonymous functions in JavaScript, you can implement entire applications based on an event-driven architecture. The term *event-driven* means that objects don't communicate directly with each other via function calls; instead, events are used for this communication. Event-driven programming is therefore primarily used to control the program flow. In contrast to the classical approach, where the source code is run through linearly, here functions are executed when certain events occur. A small example in Listing 1.2 illustrates this approach.

```
myObj.on('myEvent', (data) => {
  console.log(data);
});
myObj.emit('myEvent', 'Hello World');
```

Listing 1.2 Event-Driven Development in Node.js

You can use the `on` method of an object that you derive from `events.EventEmitter`, a component of the Node.js platform, to define which function you want to use to

respond to each event. This pattern is referred to as a publish-subscribe pattern. Objects can thus register with an event emitter and then be notified when the event occurs. The first argument of the `on` method is the name of the event in the form of a string to respond to. The second argument consists of a callback function that is implemented as an arrow function in this case, which is executed once the event occurs. Thus, the function call of the `on` method does nothing more than register the callback function the first time it's executed. Later in the script, the `emit` method is called on `myObj`. This ensures that all callback functions registered by the `on` method are executed.

What works in this example with a custom object is used by Node.js to perform a variety of asynchronous tasks. However, the callback functions aren't run in parallel, but sequentially. The single-threaded approach of Node.js creates the problem that only one operation can be executed at a time. Time-consuming read or write operations in particular would block the entire execution of the application. For this reason, all read and write operations are outsourced using the event loop. This allows the available thread to be exploited by the application's code. Once a request is made to an external resource in the source code, it's passed to the event loop. A callback is registered for the request that forwards the request to the operating system; Node.js then regains control and can continue executing the application. Once the external operation is complete, the result is passed back to the event loop. An event occurs and the event loop ensures that the associated callback functions are executed. Figure 1.4 shows how the event loop works.

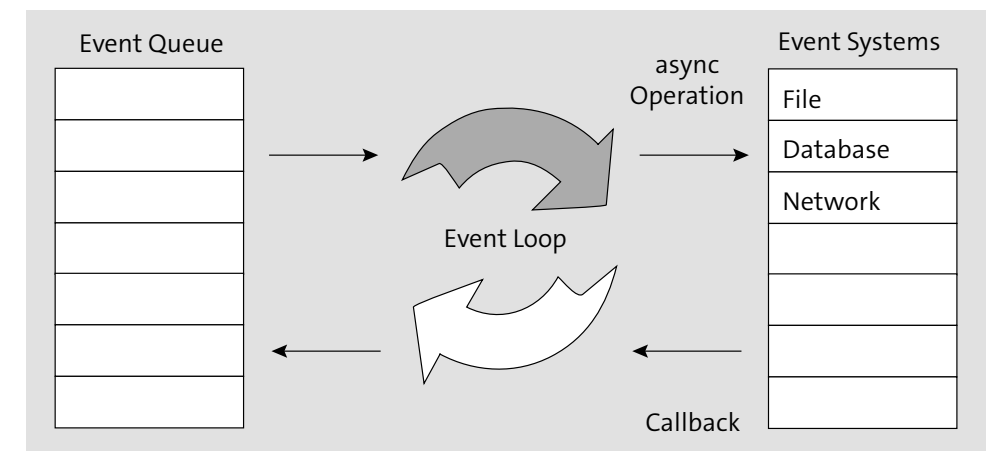


Figure 1.4 Event Loop

The original event loop used in Node.js is based on `libev`, a library written in C that stands for high performance and a wide range of features. `libev` is based on the approaches of `libevent` but has a higher performance rate, as evidenced by various benchmarks. Even an improved version of `libevent`—`libevent2`—doesn't match the performance of `libev`. However, for compatibility reasons, the event loop was abstracted to achieve better portability to other platforms.

1.7.2 Input and Output

The event loop alone in combination with the V8 engine allows the execution of JavaScript, but there is still no possibility of interacting with the operating system directly in the form of read or write operations on the file system. In the implementation of server-side applications, accesses to the file system play an important role. For example, the configuration of an application is often outsourced to a separate configuration file. This configuration must be read by the application from the file system. However, templates, which are dynamically filled with values and then sent to the client, are also usually available as separate files. Both reading and writing information to files is often a requirement for a server-side JavaScript application. Logging within an application is another common area of usage of write accesses to the file system. Here, different types of events within the application are logged to a log file. Depending on where the application is executed, only fatal errors, warnings, or even runtime information is written. Write accesses are also used for persisting information. During runtime of an application, usually through the interaction of users and various computations, information is generated that needs to be captured for later reuse.

Node.js uses the C library `libeio` for these tasks. It ensures that the write and read operations can take place asynchronously, and thus the library works very closely with the event loop. However, the features of `libeio` aren't limited to write and read access to the file system; rather, they offer considerably more possibilities to interact with the file system. These options range from reading file information (e.g., size, creation date, or access date) to managing directories (i.e., creating or removing them) to modifying access rights. Similar to the event loop, during the course of its development, this library was separated from the actual application by an abstraction layer.

To access the file system, Node.js provides its own module, the file system module. This module enables you to address the interfaces of `libeio` and thus represents a very lightweight wrapper around `libeio`.

1.7.3 libuv

The two libraries you've encountered so far are related to Linux. However, Node.js was supposed to become a platform independent of the operating system. For this reason, the `libuv` library was introduced in version 0.6 of Node.js. This library is primarily used to abstract differences between different operating systems. Consequently, using `libuv` makes it possible for Node.js to run on Windows systems as well. The structure without `libuv`, as it was used in Node.js up to version 0.6, looks like this: the core is the V8 engine; it's supplemented by `libev` and `libeio` with the event loop and the asynchronous file system access. With `libuv`, these two libraries are no longer directly integrated into the platform, but are abstracted.

For Node.js to work on Windows, it's necessary to provide the core components for Windows platforms. The V8 engine isn't a problem here; it has been working in the Chrome

browser for many years on Windows without any problems. However, it gets more difficult with the event loop and asynchronous file system operations. Some components of `libev` would need to be rewritten when running on Windows. In addition, `libev` is based on native implementations of the operating system of the `select` function, but, on Windows, a variant optimized for the operating system is available in the form of IOCP. To avoid having to create different versions of Node.js for the different operating systems, the developers decided to include an abstraction layer with `libuv` that allows `libev` to be used for Linux systems and IOCP for Windows. With `libuv`, some core concepts of Node.js have been adapted. For example, we no longer speak of events, but of operations. An operation is passed to the `libuv` component; within `libuv`, the operation is passed to the underlying infrastructure, that is, `libev` or IOCP, respectively. Thus, the Node.js interface remains unchanged regardless of the operating system used.

`libuv` is responsible for managing all asynchronous I/O operations. This means that all access to the file system, whether read or write access, is performed via `libuv`'s interfaces. For this purpose, `libuv` provides the `uv_fs_` functions, as well as timers, that is, time-dependent calls, and asynchronous Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) connections run via `libuv`. In addition to these basic functionalities, `libuv` manages complex features such as creating and spawning child processes and thread pool scheduling, an abstraction that allows tasks to be completed in separate threads and callbacks to be bound to them. Using an abstraction layer such as `libuv` is an important building block for the wider adoption of Node.js and makes the platform a little less dependent on the system.

1.7.4 Domain Name System

The roots of Node.js can be found on the internet. When you're on the internet, you'll quickly encounter the problem of name resolution. Actually, all servers on the internet are addressed by their IP address. In Internet Protocol version 4 (IPv4), the address is a 32-bit number represented in four blocks of 8 bits each. In IPv6, the addresses have a size of 128 bits and are divided into eight blocks of hexadecimal numbers. You rarely want to work directly with these cryptic addresses, especially if a dynamic assignment via Dynamic Host Configuration Protocol (DHCP) is added. The solution to this is the Domain Name System (DNS). The DNS is a service for name resolution on the web that ensures domain names are converted into IP addresses. There is also the possibility of reverse resolution, where an IP address is translated into a domain name. If you want to connect a web service or read a webpage in your Node.js application, DNS is used here as well.

Internally, Node.js doesn't handle the name resolution itself but passes the respective requests to the C-Ares library. This applies to all methods of the `dns` module except for `dns.lookup`, which uses the operating system's own `getaddrinfo` function. This exception is caused by the fact that `getaddrinfo` is more constant in its responses than the C-Ares library, which, by itself, is a lot more performant than `getaddrinfo`.

1.7.5 Crypto

The crypto component of the Node.js platform provides you with several encryption options for development purposes. This component is based on OpenSSL. This means that this software must be installed on your system if you want to encrypt data. The crypto module allows you to encrypt data with different algorithms as well as create digital signatures within your application. The entire system is based on private and public keys. The private key, as the name implies, is for you and your application only. The public key is available to your communication partners. If content is to be encrypted, this is done with the public key. The data can then only be decrypted with your private key. The same applies to the digital signature of data. Here, your private key is used to generate such a signature. The recipient of a message can then use the signature and your public key to determine whether the message originated from you and hasn't been changed.

1.7.6 Zlib

When creating web applications, as a developer, you need to take into consideration the resources of your users and your own server environment. For example, the available bandwidth or free memory for data can be a limitation. To address such cases, the Node.js platform contains the `zlib` component. With its help, you can compress data and decompress it again when you want to process it. For data compression, you can use two algorithms, Deflate and Gzip. Node.js treats the data that serves as input to the algorithms as streams.

Node.js doesn't implement the compression algorithms itself, but instead uses the established `zlib` and passes the requests on in each case. The `zlib` module of Node.js simply provides a lightweight wrapper for the underlying Gzip, Deflate/Inflate, and Brotli algorithms and ensures that I/O streams are handled correctly.

1.7.7 HTTP Parser

As a platform for web applications, Node.js must be able to handle not only streams, compressed data, and encryption but also HTTP. Because parsing HTTP is a laborious procedure, the HTTP parser handling this task has been outsourced to a separate project and is now included by the Node.js platform. Like the other external libraries, the HTTP parser is written in C and serves as a high-performance tool that reads both HTTP requests and responses. As a developer, this means you can use the HTTP parser to read, for example, the various information in the HTTP header or the text of the message itself.

The primary goal of developing Node.js is to provide a performant platform for web applications. To meet this requirement, Node.js is built on a modular structure. This allows the inclusion of external libraries such as the previously described `libuv` or the

HTTP parser. The modular approach continues through the internal modules of the Node.js platform and extends to the extensions you create for your own application.

Throughout this book, you'll learn about the different capabilities and technologies that the Node.js platform provides for developing your own applications. We'll start with an introduction to the module system of Node.js.

1.8 Summary

For many years now, Node.js has been an integral part of web development. In this context, Node.js isn't just used to create server applications but also is the basis for a wide range of tools—from the bundler webpack to tools such as Babel and the compiler for CSS preprocessors. The success of the platform is based on several very simple concepts. The platform is based on a collection of established libraries, which together create a very flexible working environment. Over the years, the core of the platform has always been kept compact, offering only a set of basic functionalities. For all other requirements, you can use the npm to integrate a wide variety of packages into your application.

Although Node.js has now been proven in practice for several years, you may still frequently hear the following question: Can I safely use Node.js for my application? In the versions prior to 0.6, this question could not be answered in the affirmative in good conscience because the interfaces of the platform were subject to frequent changes. Today, however, Node.js is much more mature. The interfaces are kept stable by the developers. The LTS version was created for use in enterprises. This is a Node.js version that is supported by updates for a total of 30 months. This increases the reliability of the platform and takes the pressure off companies to always update to the latest version.

A thoroughly exciting chapter in the development history was the separation of `io.js` because the development of Node.js had lost its momentum, and no innovations entered the platform for a long time. This event was a crucial turning point for the development of Node.js. The Node.js Foundation was formed, and responsibility for development was transferred from an individual to a group. As a result, release cycles and versioning were standardized, signaling both reliability and continuous further development to users of the platform.

By deciding to delve deeper into Node.js, you'll be in good company with numerous enterprises large and small around the world that are now strategically using Node.js for application development.

Chapter 6

Express

The future was better in the past too!

– Karl Valentin

Express has been the most popular web application framework for Node.js for many years. The open-source project was launched in June 2009 by T. J. Holowaychuk, and its purpose is to help you develop web applications. The focus of Express is on speed, manageable scope of the core framework, and an easily extensible interface. The well-thought-out architecture makes it possible to maintain until today, and thus the framework has become an almost indispensable companion when it comes to the development of web server applications based on Node.js. Frameworks such as Express exist because web development often involves solving standard tasks. For example, in PHP, there is the Symfony framework; in Python, you can use Django; and Ruby on Rails offers a solution for web applications under Ruby. You can implement your application completely in the respective language (in this case, Node.js) without the help of further libraries and frameworks, but you lose a lot of time with the implementation of the basic infrastructure. Just remember the `createServer` callback function from the previous chapter where you had to take care of parsing the URL and performing the corresponding action yourself.

In addition to handling requests and resolving URLs, other standard tasks such as session handling, authentication, or file uploads need to be taken care of. There are already established solutions for all these tasks, which have been combined into a framework under the leadership of Express. Because of its stability over the years and its extensible architecture, Express serves as the foundation for a variety of other libraries and frameworks, such as Nest, which we'll look at in more detail in Chapter 14.

6.1 Structure

Express is a compact framework with a manageable range of functions. However, it can be easily extended with middleware components. The structure of Express, much like Node.js itself, is layered, as you can see in Figure 6.1.

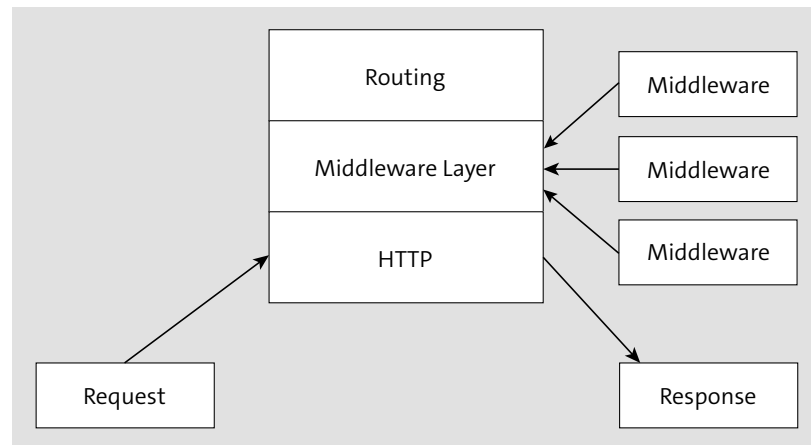


Figure 6.1 Express Structure

The `http` module of Node.js serves as the foundation for Express. The `http` module creates the server process on which Express is based. In addition, the `request` and `response` objects are available to access the request information and create the response to the client, respectively. Internally, for example, Express makes use of the URL the user has entered in the browser to implement the routing process within the application.

The second layer of the Express architecture is the middleware layer. In the context of Express, a middleware is a function located between the incoming request and the server's response to the client. Multiple middleware functions can be chained together to perform specific actions based on the client's request. Prior to the third version of Express, Connect formed this middleware layer. In version 4, the developers abandoned this additional dependency and developed a standalone layer, which remains mostly compatible. The third layer of the Express architecture is the router. This component of Express controls which function should be executed depending on the called URL to generate a response to the client. When routing, both the HTTP method and the URL path are considered.

In this chapter, you'll create a web application that allows you to manage a movie database. Before you start working on the application, you must first initialize it and install Express.

6.2 Installation

Express follows the Node Package Manager (npm) package standard: It's freely available as an open-source project, subject to the MIT License and developed on GitHub. The Express package is available through a package manager of your choice, for example, npm. Before installing Express in your application, you must use `npm init -y` on the command line to generate the `package.json` file for your application and add the type

field with the `module` value to use the ECMAScript module system. Then you must install Express via the `npm install express` command. After that, you can test the functionality of the framework with a simple application. Listing 6.1 shows the source code of this first step. You save the source code in a file named `index.js`.

```
import express from 'express';

const app = express();

app.get('/', (req, res) => {
  res.send('My first express application');
});

app.listen(8080, () => {
  console.log('Movie database accessible at http://localhost:8080');
});
```

Listing 6.1 First Express Application (`index.js`)

In the first step, you include the `express` package. The default export of the package is a function that you use to create the base for your application via the `app` object. The `get` method of the `app` object creates a route, in this case, for the path `/`. In the last step, you bind your application to Transmission Control Protocol (TCP) port 8080. Internally, a server is created at this point using the `http` module of Node.js and bound to the specified port. Once you've started the application via the `node index.js` command, you can access and test it in the browser using the URL `http://localhost:8080`. The results are displayed in Figure 6.2.

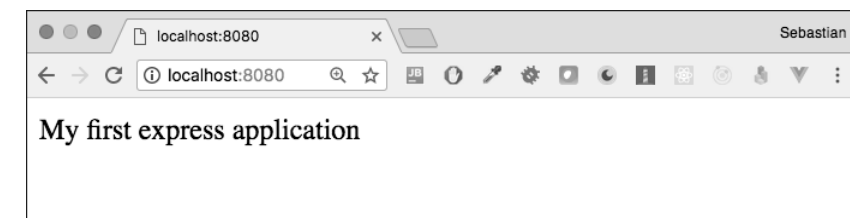


Figure 6.2 Output of the Express Application in the Browser

Based on this working foundation, you can now proceed to extend this example step by step to a fully functional application.

6.3 Basic Principles

The process by which an Express application works always has the same pattern. The Express server receives a request from a client. Based on the chosen `http` method and

URL path, a suitable route is chosen, and one or more callback functions are executed. Within this callback function, you can access the `request` and `response` objects, as is the case with the `http` module. These two objects, along with the router and middleware components, are the heart of an application.

6.3.1 Request

The `request` object is the first argument of the Express routing callback functions and represents the user’s request. You can extend this object by using middleware components such as the body parser and the cookie parser, for example, so that you can manage cookies or other aspects of the communication with the client more conveniently. But even in standard mode, it contains a lot of helpful information. Table 6.1 introduces you to some of the most important properties of the `request` object.

Characteristic	Description
method	Contains the HTTP method used to send the request to the server.
originalUrl	Contains the original request URL. This allows the <code>url</code> property, which contains the same information, to be modified for in-application purposes.
params	Contains the value that consists of the variable parts of the URL. You’ll learn how to define and use them in Express later in this chapter.
path	Enables you to access the URL path.
protocol	Contains the protocol of the request, such as HTTP or HTTPS.
query	Accesses the query string is a part of the URL.

Table 6.1 Key Properties of the “request” Object

In addition to the properties, you also have access to some methods that allow you to read more information about the incoming request. The `get` method enables you to read header fields from the request. For example, if you’re interested in the `Content-Type` field, the call is `req.get('Content-Type')`. It doesn’t matter whether you use uppercase or lowercase letters with this method.

6.3.2 Response

The `response` object, which you can access via the second argument of the routing function, represents the response to the client. Because you mainly write to this object, it also provides significantly more methods than properties. The most important property of the object is `headersSent`. This Boolean value tells you whether the HTTP headers of the response have already been sent. If this is the case, you can no longer modify it. Table 6.2 provides an overview of the most important methods of the `response` object.

method	Description
<code>get(field)</code>	Reads the specified header field of the response.
<code>set(field[, value])</code>	Sets the value of the specified header field.
<code>cookie(name, value[, options])</code>	Sets a cookie value.
<code>redirect([status,]path)</code>	Forwards the request.
<code>status(code)</code>	Sets the status code of the response.
<code>send([body])</code>	Sends the HTTP response.
<code>Json([body])</code>	Sends the HTTP response. The passed object is converted to a JavaScript Object Notation (JSON) object, and the correct response headers are set.
<code>end([data][, encoding])</code>	Sends the HTTP response. You should use this method primarily if you don’t send user data such as HTML structures. Otherwise, you should use the <code>send</code> method.

Table 6.2 Key Methods of the “response” Object

After this brief introduction to the request and response handling elements, the following sections deal with the setup and architecture of an Express application.

6.4 Setup

As a general best practice in dealing with Express, it has emerged that an application should be divided into separate components as far as possible, each of which is stored in separate files. Although you create a lot of files with this strategy, depending on the size of your application, you’ll still be able to locate the files quickly due to a well-structured directory hierarchy. A file contains only one component and thus a self-contained unit. For structuring an Express application, a classic model-view-controller (MVC) approach is the best choice.

MVC: The Model-View-Controller Pattern

The MVC pattern is used to structure applications, especially in web development, where it has become an important standard. This pattern describes where certain parts of an application should be stored and how these parts interact. The name MVC already contains the three components of the pattern:

- **Model**
The models of an MVC application are used for data management. Models encapsulate all operations related to the data of your application. This concerns both the

creation and the modification or deletion of information. Typically, a model encapsulates database accesses. In addition to the pure data and the associated logic for handling it, models also encapsulate the business logic of your application.

■ View

The task of views consists of displaying information. The views of an Express application are mostly HTML templates populated with the dynamic data of your application before delivery to the client. In modern applications that are application programming interface (API) heavy, this aspect of the MVC architecture is increasingly taking a back seat, as templates are rarely rendered, and, instead, JSON objects are often sent from the server to the client.

■ Controller

A controller contains the control logic of your application. The controller brings models and views together. You should make sure that your controllers don't become too large. If the controller contains too much logic, you should outsource it to support functions or models.

When building your application, it's critical that you follow a consistent convention when structuring the file and directory hierarchy to maintain maintainability and extensibility over the lifecycle of the application.

6.4.1 Structure of an Application

The choice of directory structure depends very much on the scope of your application. Avoid making the structure unnecessarily complex at the very start because when you start working on an application, you usually don't know exactly what it will look like in the end. The more complex the structure becomes, the more time-consuming its adjustments become. Start with as flat a hierarchy as possible, and restructure the directories and files as needed. The module system of Node.js and modern development environments support you in this continuous refactoring process by easily moving files and directories and automatically adjusting `import` statements. An application usually consists of the following components: models, views, controllers, routers, and helpers.

Structure for Small Applications

For very small applications, you should create one file per component and place it in a directory. An example of such a structure is given in Figure 6.3.

This structuring approach only works for very manageable applications or small prototypes. As soon as your project has more than three or four separate endpoints, you should switch to the next larger variant or start with it directly, as migrating the structure in this case will only become unnecessarily time-consuming.

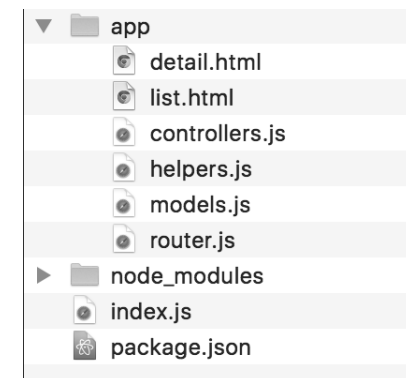


Figure 6.3 Directory Structure for Small Applications

Structure for Medium-Sized Applications

Web applications that have 10 to 15 independent endpoints, that is, separate routes, fit into the category of medium-sized applications. This structuring variant is a good starting point for normal web applications. In this structure, the different components—models, views, and controllers—are stored in separate directories. All structures are located in their own files and are named accordingly. To make it easier to locate the structures in the development environment, it has become best practice to include the type of structure in the file name. For example, a controller that is responsible for the login process of your application is then located in a file named `login.controller.js` in the `controllers` directory. The structure of such an application could look like the one shown in Figure 6.4.

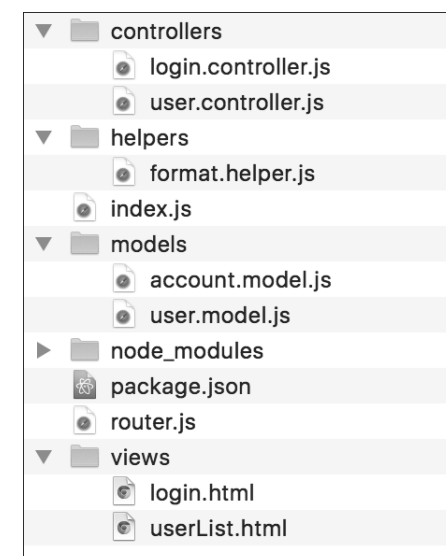


Figure 6.4 Structure for Medium-Size Applications

The advantage of this structuring variant is that all structures are stored separately from each other and also within the respective component; for example, in the case of models, a thematic separation takes place at the file level. This decouples the different areas of the application and keeps the number of files per directory low. For small- and medium-sized applications this is a very good approach.

Structure for Large Applications

As far as the scope of your application is concerned, you’re hardly limited when using Express. However, when developing the file and directory structure of large-scale applications, you should take an approach that allows you to thematically separate the modules of your application. Such a strategy allows you to develop your application in parallel with several teams and manage the individual modules independently. This also greatly simplifies the localization of code locations. Figure 6.5 shows an example of such a structure.

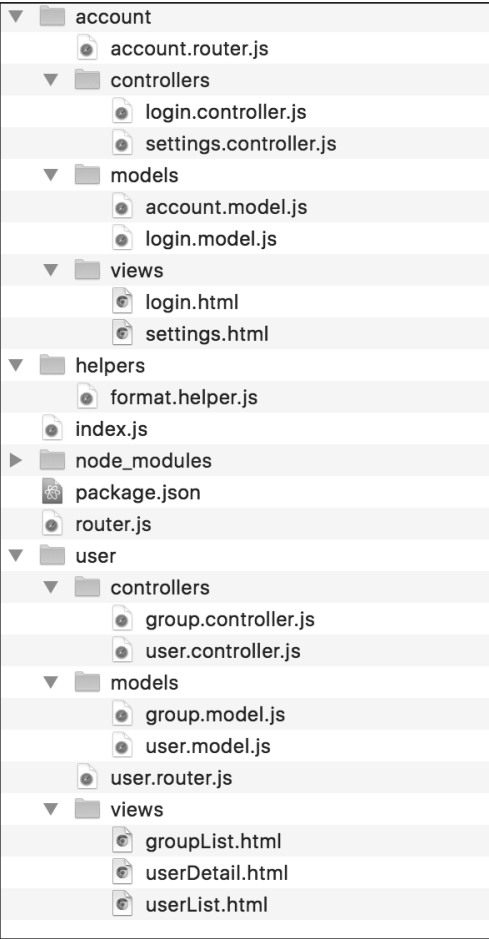


Figure 6.5 Structure for Extensive Applications

As you can see in Figure 6.5, there are separate directories for each subject area of the application. The selection and delineation of these areas depend entirely on you and the requirements of your application. Typically, modules are chosen to cover a topic in its entirety. This approach enables clean interfaces to the other parts of the application. The available options here range from logical units within an endpoint to groups of multiple thematically related endpoints. Each module in turn contains subdirectories, each of which contains the module’s models, views, and controllers. In addition, each module defines its own router. Now that you have an overview of the structuring options for your application, you can move on to extending your first sample application in the next step.

6.5 Movie Database

A movie database will serve as a sample application for Express. This covers all essential aspects of a typical application. You can add movies to the database, view the existing records, update them if necessary, and also delete them. In addition, further functionalities such as ratings can be implemented. This application also serves as a basis for the following chapters when it comes to integrating template engines, connecting different databases, or authenticating users.

For the sample application, we chose the structure of a large application, although it will initially consist of only one module. The reason is that this structure offers the highest degree of flexibility, allowing you to get to know more features of Express.

The first step in the implementation of the application consists of the initialization. For this purpose, you should create a directory named *movie-db* and go to this directory via the command line. After that, you must run the `npm init -y` command to create a *package.json* file. Once you’ve created the file, you must add the `type` field with the `module` value to be able to use the ECMAScript module system. You’re relatively free to choose a name, but you should select a unique name in case you want to publish your application. Note that the name must be written in lowercase letters and consist of only one word. However, it may contain hyphens and underscores. Then you must install Express using the `npm install express` command. Listing 6.2 shows the *package.json* file of the application.

```
{
  "name": "node-book",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "start": "node index.js"
  },
}
```

```

"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
  "express": "^4.17.1"
}
}

```

Listing 6.2 Initial Configuration of the Application (package.json)

The two changes you need to make manually to the file are the `type` field and a startup script that allows you to conveniently start your application using the `npm start` command.

In the final step of initialization, you must create an *index.js* file in the root directory of your application as the entry point to your application. During development, you should make sure that this file is only responsible for initializing the application and doesn't perform any other tasks. The preliminary source code of this file is shown in Listing 6.3.

```

import express from 'express';

const app = express();

app.listen(8080, () => {
  console.log('Server is listening to http://localhost:8080');
});

```

Listing 6.3 Getting Started with the Application (index.js)

6.5.1 Routing

In the first step, you want your application to output a simple list of movie titles. For this purpose, you must first define a module, that is, a unit in your application that covers all aspects related to movies in the application. Based on the default guidelines for the structure, you first need a directory with the name of the module, in this case, *movie*. In this directory, you create a file named *index.js*. The file is the entry point to the module. If you integrate it into your application at a later point in time, this file ensures that all other relevant parts of the module are loaded so that the integration causes as little effort as possible.

The index file of your module exports the `router` object you later include in the index file of your application. In addition to creating the `router` object, this file currently takes care of managing the data and the response to the request in the callback function of the route. Listing 6.4 contains the source code of the file.

```

import { Router } from 'express';

const router = Router();

const data = [
  { id: 1, title: 'Iron Man', year: '2008' },
  { id: 2, title: 'Thor', year: '2011' },
  { id: 3, title: 'Captain America', year: '2011' },
];

router.get('/', (request, response) => {
  response.send(data);
});

export { router };

```

Listing 6.4 Router File of the Movie Module (movie/index.js)

Before you can display the data in the browser, you must integrate the router into your application. This is done in the *index.js* file in the root directory of your application, as shown in Listing 6.5.

```

import express from 'express';
import { router as movieRouter } from './movie/index.js';

const app = express();

app.use('/movie', movieRouter);

app.get('/', (request, response) => response.redirect('/movie'));

app.listen(8080, () => {
  console.log('Server is listening to http://localhost:8080');
});

```

Listing 6.5 Integrating the Router (index.js)

When loading the router, you must specify the name of the file in the ECMAScript module system, that is, *index.js*, as Node.js doesn't use this automatically when specifying a directory. This is different from the CommonJS module system. To make the code a bit more meaningful, you should rename the `router` to `movieRouter` on import. The `use` method specifies that the `movieRouter` is responsible for the `/movie` path. Currently, users of your application need to know that the movie list can be found at *http://localhost:8080/movie*. By using the `get` route for the `/` path and then redirecting to `/movie`, you enable users to also access the list via *http://localhost:8080*. This way, you've defined the entry point for your application. If you start your application via the `npm`

start or node index.js commands, you can open it in your browser and get a display similar to the one shown in Figure 6.6.

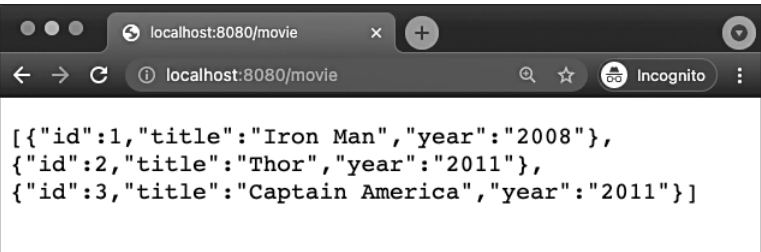


Figure 6.6 Movie List in the Browser

Patterns in Routes

Static routes, as you’ve come to know them so far, cover most use cases in a web application. However, there are use cases where you reach the limits due to little flexibility. For this reason, it’s possible to formulate dynamic routes in Express.

Patterns	Example	Path	Description
?	/ab?c	/abc or /ac	Sign may occur, but doesn’t have to.
+	/ab+c	/abc or /abbc	Character occurs once or multiple times.
*	/a*c	/ac or /aBCc	Any character string.

Table 6.3 Patterns in Routes

In addition to the patterns listed in Table 6.3, you can create groups of characters by means of parentheses and apply the multipliers to these groups. Thus, a route from /a(bc)?d applies to both /ad and /abcd. If these options are still not sufficient to cover your use case, you can also specify routes as regular expressions. Listing 6.6 shows an example of such a route, which is responsible for all paths that contain the string /movie somewhere in the path.

Route Dependency

Note that a route in Express can cause other routes to stop running, depending on where you place it in your application. Express always uses the first suitable route it can find. If it sends its response to the client, the middleware chain is broken, and Express doesn’t perform any further functions for this request.

```
app.get(/.*\movie.*$/, function (request, response) {
  response.send('Movie Route');
});
```

Listing 6.6 Regular Expressions as Routes

If you formulate your routes with regular expressions, you have maximum flexibility. However, this often makes the routes less legible, which makes troubleshooting more difficult. For this reason, you should use regular expressions sparingly in this case and rather as an exception when the normal route definitions are no longer sufficient.

6.5.2 Controller

At this point, your router still implements the complete MVC pattern on its own. This isn’t desirable because readability suffers with increasing functionality, so in the next step, you must outsource everything except the actual route definition to a controller. The controller has the task of merging the view and the model. In addition, at this point, the information is usually extracted from the request, and the response to the client is formulated. As to the naming of routing callback functions, it has become standard in many web frameworks to refer to these functions as actions. A controller can contain several of these action functions. Listing 6.7 shows the implementation of the controller for the list view.

```
const data = [
  { id: 1, title: 'Iron Man', year: '2008' },
  { id: 2, title: 'Thor', year: '2011' },
  { id: 3, title: 'Captain America', year: '2011' },
];

export function listAction(request, response) {
  response.send(data);
}
```

Listing 6.7 Controller (movie/controller.js)

Export and implementation are linked in this case. However, at this point, you may as well collect all exports at the end of the file. There is no right or wrong here; the only important thing is to stay consistent and always use the same type of export. Typically, you use the combination of export and definition in files that contain only a few exports. In general, you should make sure that a file doesn’t export too many structures, as this can quickly become confusing and indicate that the file hosts too many structures.

The controller is included in the router, where a reference to the listAction method of the controller is entered instead of the routing callback function. The source code of the customized router is shown in Listing 6.8.

```
import { Router } from 'express';
import { listAction } from './controller.js';

const router = Router();
```



```
router.get('/', listAction);
```

```
export { router };
```

Listing 6.8 Including the Controller Action in the Router (movie/index.js)

When extending the router, you must import `listAction` and use it in the `get` method of the router. With this modification, you've separated the routing of your module and the handling of request and response. However, model, view, and controller are still quite tightly connected. The next step is to resolve this.

6.5.3 Model

The movie database data has the form of a simple array of objects. The model encapsulates this array and provides a function to read the data and later further methods to modify this data structure. You're not yet working with a database or other external system for data storage in your application. But to make the source code of the application a bit more realistic, you should implement the interfaces of the model with promises and, thus, asynchronously.

Promises

A *promise* is an object that represents the fulfillment of an asynchronous operation in JavaScript. Unlike callback functions, you can work much better with promises and also bind multiple operations to the fulfillment of such a promise object.

Promises are a language feature of JavaScript and part of the standard, so they are supported on both the client side and server side. You can create a promise object either with the `promise` constructor or with `Promise.resolve` or `Promise.reject`. Listing 6.9 shows a simple example of a function that works with promises.

```
function asyncFunction() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Hello world!');
    }, 1000);
  });
}
```

```
const promise = asyncFunction();
promise.then((value) => {
  console.log(value);
});
```

Listing 6.9 Using Promises

You pass a callback function to the promise constructor in the `asyncFunction` function. This function has access to the `resolve` and `reject` arguments. You can use these functions to indicate a success or failure of an asynchronous operation by calling the respective function. Here, you can pass any value that represents something like the return value of the asynchronous operation.

The promise object that the `asyncFunction` returns has the `then`, `catch`, and `finally` methods. Each of these methods accepts a callback function that will be executed in case of success, error, or both, respectively, and will be passed the value you passed when calling `resolve` or `reject`. The `then` method is a special case because here you can specify a second callback function for the case of an error.

The data of the model is in the scope of the file and can't be modified directly from outside. The `getAll` method returns a promise object that you can create using the `Promise.resolve` method (see Listing 6.10). This promise object is resolved with a reference to the data. This is a problem at first because the information could be changed via this reference. At a later date, this data structure will be exchanged for a fully-fledged database, so there's currently no need to worry about this problem. After you've created the basis for the model, you still need to integrate it into your application. The controller is the place to go for this. Listing 6.11 shows the adjustments you need to make to your code.

```
const data = [
  { id: 1, title: 'Iron Man', year: '2008' },
  { id: 2, title: 'Thor', year: '2011' },
  { id: 3, title: 'Captain America', year: '2011' },
];
```

```
export function getAll() {
  return Promise.resolve(data);
}
```

Listing 6.10 Implementing the Model (movie/model.js)

```
import { getAll } from './model.js';
```

```
export async function listAction(request, response) {
  const data = await getAll();
  response.send(data);
}
```

Listing 6.11 Adapting the Controller to Integrate the Model (movie/controller.js)

If you implement the `listAction` function as an `async` function as in our example, you can use the `await` keyword within this function to wait for the promise to resolve. This whole task is done asynchronously, that is, nonblocking, making the source code much more readable than if you were using callback functions. Express supports this type of asynchronicity directly and without additional configuration.

6.5.4 View

The final component of your MVC application with Express is the view. This part is responsible for the display. The output of a JavaScript object isn't a particularly appealing form of presentation. Therefore, it's better to use HTML as the output format at this point. You have several options for the display. For example, you can save the HTML code in a separate HTML file, read it with JavaScript, and replace the appropriate sections. You can either do this replacement directly using the `replace` method of the HTML string, or you can use an HTML parser such as Cheerio. A simpler variant is to use JavaScript template strings, as you'll see later in this section. Another option is to use a template engine. The next chapter describes this topic in greater detail. Listing 6.12 contains the movie list view.

```
export function render(movies) {
  return `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Movie list</title>
</head>
<body>
  <table>
    <thead><tr><th>Id</th><th>Title</th></tr></thead>
    <tbody>
      ${movies
        .map(movie => `<tr><td>${movie.id}</td><td>${movie.title}</td></tr>`)
        .join('')}
    </tbody>
  </table>
</body>
</html>
`;
};
```

Listing 6.12 Displaying the Movie List View (movie/view.js)

The view in this case consists of a `render` function that receives the data to be displayed as an argument. Within the HTML structure, each entry is transformed into a table row using the `map` method. You connect this structure with the `join` method into a character string, which is then output at the correct position within the template string. The task of the controller is to bring the model and view together. Listing 6.13 contains the customized code of the controller.

```
import { getAll } from './model.js';
import { render } from './view.js';

export async function listAction(request, response) {
  const data = await getAll();
  const body = render(data);
  response.send(body);
}
```

Listing 6.13 View Integration in the Controller (movie/controller.js)

The `listAction` function loads the list of movies in the first step. This is passed to the view in the second step. In the last step, you send the HTML structure to the client. When you restart your application via the command line using the `npm start` command now, you'll get an output like the one shown in Figure 6.7.

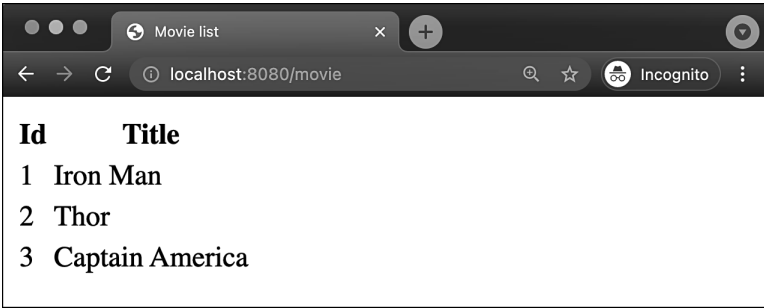


Figure 6.7 Display of the Movie List after the View Conversion

When implementing your application, make sure that you move as little logic as possible into the view. Iteration, as you've implemented here, or simple conditions to hide or show parts are recommended options. You should definitely outsource more extensive logic to your models.

6.6 Middleware

A key design feature of Express is the middleware concept, which makes the framework a very flexible tool. Simply put, middleware here refers to a function that stands

between the incoming request and the outgoing response. You can put as many of these functions in succession as you like. There are already numerous predefined middleware functions that you can include in your application and that perform specific tasks for you. If all this doesn't provide any solution to your problem, you can even write such a function yourself.

6.6.1 Custom Middleware

You can use a middleware function to extract information from the incoming request and store it, such as with a logger. However, you can also enrich the response based on information from the request. A middleware function must have a specific signature so that it doesn't break the chain of middleware functions. Suppose you wanted to define a logger for your application that writes each request to the console. To do this, you must adjust the contents of your application's initial file according to Listing 6.14.

```
import express from 'express';
import { router as movieRouter } from './movie/index.js';

const app = express();

function log(request, response, next) {
  console.log(request.url);
  next();
}
app.use(log);

app.use('/movie', movieRouter);

app.get('/', (request, response) => response.redirect('/movie'));

app.listen(8080, () => {
  console.log('Server is listening to http://localhost:8080');
});
```

Listing 6.14 A Simple Logger (index.js)

As you can see in Listing 6.14, the signature of a middleware function is similar to that of an ordinary routing function. A middleware function always has the three parameters: request, response, and a callback function. The request and response parameters provide access to the request and response, just like in the controller actions. The request object is used in the example to output the requested URL to the console. The passed callback function, which by convention is named `next`, ensures that the next middleware function in the chain is called when the function is called. If you don't perform this callback function, usually no response is sent to the client, and the client

receives a time-out error. You can register your middleware using the `use` method of your application. As the first argument, this method optionally accepts a URL path to which the middleware should be applied. If you only pass a callback function, the middleware will be executed on every request. The order in which you register your functions is important. If you first register a regular routing method that doesn't call the next callback function, your middleware component, which is registered afterwards, won't be executed. If your middleware is to perform calculations or modifications you need later in the call chain of the functions, you can cache this information in the request object or, better yet, in the response object within a property. These objects are available to all functions as a reference and can therefore also be used to transport information.

Especially for standard tasks, such as the creation of an access log, there are prefabricated components that you only have to install and integrate. For a list of middleware components, see <http://expressjs.com/en/resources/middleware.html>.

6.6.2 Morgan: Logging Middleware for Express

Logging incoming requests is a standard problem for which there are established solutions. Morgan is one of the most popular middleware components available that does this work for you. Use the `npm install morgan` command to install the package in your application. At its core, Morgan consists of a function that accepts a format for the log entries and additional options. This means you can replace your current logger implementation with Morgan in the next step, as shown in Listing 6.15.

```
import express from 'express';
import morgan from 'morgan';
import { router as movieRouter } from './movie/index.js';

const app = express();

app.use(morgan('common', { immediate: true }));

app.use('/movie', movieRouter);

app.get('/', (request, response) => response.redirect('/movie'));

app.listen(8080, () => {
  console.log('Server is listening to http://localhost:8080');
});
```

Listing 6.15 Morgan Middleware

Concerning the format, you can choose from a number of predefined formats such as `combined`, `short`, or `dev`. The `common` format used in the example is similar to the format

used by the Apache web server in the access log. As an alternative to the predefined formats, you can also define a format yourself. For example, if you want to record only the date, HTTP method, URL, and status code, the corresponding format looks like this: `':date :method :url :status'`. The third variant for defining a format consists of using a function instead of a character string. Irrespective of the variant you decide on, you must pass it to the `morgan` function as the first argument. The `option` object, which you pass as the second argument to Morgan, allows you to manipulate the behavior of the logger. With the `immediate` key used in the example, you specify whether the log entry should be written immediately or only when the response is sent to the client. The `stream` property allows you to specify a writable stream to which the log entries are written. This allows you to write the log entries to not only the console but also a file.

In Listing 6.16, you can see how to use the `createWriteStream` function from the `fs` module to open a data stream that writes to the `access.log` file and pass it to Morgan using the `stream` property of the configuration object. This results in a new entry being written to the file each time it's accessed. The configuration object of the `createWriteStream` function with the `flags` property and the `a` value ensures that new entries are appended and the existing content isn't overwritten. Furthermore, the file gets created, if it doesn't already exist.

The `skip` property allows you to specify a function that can access the request and response and whose return value determines whether an entry is written to the log or not. If the `skip` function returns the `false` value, the entry won't be written.

```
import express from 'express';
import morgan from 'morgan';
import { createWriteStream } from 'fs';
import { router as movieRouter } from './movie/index.js';

const app = express();

const accessLogStream = createWriteStream('access.log', { flags: 'a' });
app.use(morgan('common', {
  immediate: true,
  stream: accessLogStream
}));

app.use('/movie', movieRouter);

app.get('/', (request, response) => response.redirect('/movie'));

app.listen(8080, () => {
  console.log('Server is listening to http://localhost:8080');
});
```

Listing 6.16 Writing Log Entries to a File (index.js)

6.6.3 Delivering Static Content

Web applications that you implement with Express usually not only consist of dynamic content but also require static files. For this reason, HTML, JavaScript, CSS, and image files must be loaded. Although you can use the `fs` module to read the contents of these files and send them to the client as a response, this task becomes much easier if you use the `static` middleware. Unlike Morgan, that is a component of Express, so you don't need to install any additional packages. If you take a look at the movie list in your browser, you won't be presented with a particularly attractive sight. However, this can be changed with a little bit of CSS. The CSS code shown in Listing 6.17 makes the table look slightly more appealing.

```
table {
  border-spacing: 0
}
th {
  background-color: black;
  font-weight: bold;
  color: lightgrey;
  text-align: left;
  padding: 5px;
}
td {
  border-top: 1px solid darkgrey;
  padding: 5px;
}
tbody tr:hover {
  background-color: lightgrey;
}
```

Listing 6.17 Styling the List (public/style.css)

To apply the styling to the movie list, you must adjust your application in two places. First, you need to make sure that the server delivers the CSS file and that the browser actually loads it. The delivery is carried out via the already mentioned `static` middleware. As was the case with the logger or the router, you include the `static` middleware via the `use` method of the `app` object. Listing 6.18 shows the customized initial file of your application.

```
import express from 'express';
import morgan from 'morgan';
import { dirname } from 'path';
import { fileURLToPath } from 'url';
import { router as movieRouter } from './movie/index.js';
```



```
const app = express();

app.use(express.static(`${dirname(fileURLToPath(import.meta.url))}/public`));

app.use(morgan('common', { immediate: true }));

app.use('/movie', movieRouter);

app.get('/', (request, response) => response.redirect('/movie'));

app.listen(8080, () => {
  console.log('Server is listening to http://localhost:8080');
});
```

Listing 6.18 Integrating the “static” Middleware (index.js)

When you call the `static` middleware, you pass the name of the directory where the static content resides. For the movie list, this is the *public* directory in the root of your application. In the next step, you can now reference the CSS file via a `link` tag in the HTML code of the list, as shown in Listing 6.19.

```
export function render(movies) {
  return `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Movie list</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <table>...</table>
</body>
</html>
`;
}
```

Listing 6.19 Embedding the CSS File into the HTML Code (movie/view.js)

If you modify the list view source code according to Listing 6.19 and restart your application, you’ll get a view similar to the one shown in Figure 6.8.

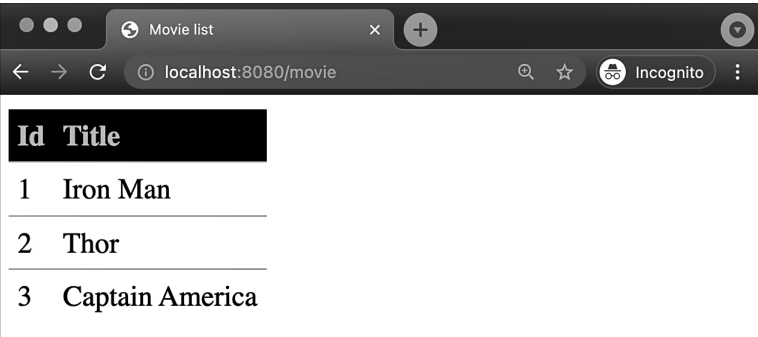


Figure 6.8 List View with CSS

6.7 Extended Routing: Deleting Data Records

Until now, the functionality of your application is limited to the display of data. In the next step, you provide your users with the option to delete data, as shown in Listing 6.20. To do this, you must first insert one link per record in the frontend. This link leads to a route on the server that takes care of deleting the record. After the deletion is done, you must redirect the user to the list so that the view gets updated.

```
<table>
  <thead><tr><th>Id</th><th>Title</th><th></th></tr></thead>
  <tbody>
    ${movies
      .map(
        movie => `
          <tr>
            <td>${movie.id}</td>
            <td>${movie.title}</td>
            <td><a href="/movie/delete/${movie.id}">delete</a></td>
          </tr>`,
        )
      .join('')}
  </tbody>
</table>
```

Listing 6.20 Extending the Template with a Delete Link (movie/view.js)

The routing functions in Express support all available HTTP methods, including, for example, the `DELETE` method. This should normally be the preferred way of deleting data, as each HTTP method has a specific meaning. In classic web applications, however, the only method you can invoke through a link is `GET`. For this reason, you must add an appropriate `get` route to your router for deleting data records. In Chapter 10,

when you implement a representational state transfer (REST) server, the situation is different. When you take a look at the delivered source code, you'll notice that each link in the table looks different because you specify the ID of the record you want to delete. This leads to a peculiarity in specifying the path in the router link. In this case, the link contains a variable portion representing the ID you can access through the `request` object. You must mark such a variable in the specification of the path of the routing function by a colon followed by the name of the variable. Listing 6.21 shows the modification at the router.

```
import { Router } from 'express';
import { listAction, removeAction } from './controller.js';

const router = Router();

router.get('/', listAction);
router.get('/delete/:id', removeAction);

export { router };
```

Listing 6.21 Extending the Router with a Delete Route (movie/index.js)

In the `removeAction` of the controller, you can access the variables of the route via the `params` object of the request. In this case, you can reach the ID passed by the user via `request.params.id`. Here you should note that Express interprets the transmitted values as strings. To be able to continue using the ID later, you should convert it to a number via the `parseInt` function. Listing 6.22 contains the source code of the controller.

```
import { getAll, remove } from './model.js';
import { render } from './view.js';

export async function listAction(request, response) {...}

export async function removeAction(request, response) {
  const id = parseInt(request.params.id, 10);
  await remove(id);
  response.redirect(request.baseUrl);
}
```

Listing 6.22 “removeAction” of the Controller (movie/controller.js)

Once you've converted the ID, you can call the `remove` method of the model, which will make sure that the corresponding record in the data source gets deleted. This operation is also asynchronous and works with promises, so put `async/await` here. After the operation is complete, you must redirect the user to the list using the `response.redirect` method. Here you can see another feature of the modular structure of an application:

instead of redirecting directly to `"/movie/"` in the `redirect` method, the `baseUrl` property of the `request` object is used in this case. The reason for this is that the `movie` module itself doesn't know the base URL for which it's responsible. To avoid a tight coupling between the embedding location and the module here, you must use the `baseUrl` property that contains the information. This allows you to change the `baseUrl` in the `index.js` file at a later stage without having to modify the module.

Finally, the `remove` method of the model filters out the data record to be deleted from the data source and overwrites the data source with the updated information. Listing 6.23 shows the corresponding implementation.

```
let data = [...];

export function getAll() {
  return Promise.resolve(data);
}

export function remove(id) {
  data = data.filter(movie => movie.id !== id);
  return Promise.resolve();
}
```

Listing 6.23 Customization on the Model (movie/model.js)

Note that in the model implementation, to clear the data, you must change the data array from a constant to a variable via `let`.

6.8 Creating and Editing Data Records: Body Parser

You can only transmit a few data records via variables in the URL, and especially for forms this isn't a practicable solution. As a rule, data is also sent using HTTP POST. You can benefit from this fact when extending your application. In the next step, you'll implement a way to create new records and edit existing ones. You start this customization again in the frontend of your application by adding a link to the list display, which allows your users to create new records. Furthermore, you should add a link in the table for each entry to edit the records. Listing 6.24 contains the source code of the view.

```
export function render(movies) {
  return `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
<title>Movie list</title>
<link rel="stylesheet" href="style.css" />
</head>
<body>
  <table>
    <thead><tr><th>Id</th><th>Title</th><th></th><th></th></tr></thead>
    <tbody>
      ${movies
        .map(
          (movie) => `
            <tr>
              <td>${movie.id}</td>
              <td>${movie.title}</td>
              <td><a href="/movie/delete/${movie.id}">delete</a></td>
              <td><a href="/movie/form/${movie.id}">edit</a></td>
            </tr>`,
          )
        .join('')}
    </tbody>
  </table>
  <a href="/movie/form">new</a>
</body>
</html>
`;
}
```

Listing 6.24 Customizing the View (movie/view.js)

Both links of the view point to the same route, once to `/movie/form` and once to `/movie/form/:id`. In Express, you can define optional parameters. If you attach a question mark to the parameter, it will be marked as optional, and you won't have to define two separate routes. You can see the updated router configuration in Listing 6.25.

```
import { Router } from 'express';
import { listAction, removeAction, formAction } from './controller.js';

const router = Router();

router.get('/', listAction);
router.get('/delete/:id', removeAction);
router.get('/form/:id?', formAction);

export { router };
```

Listing 6.25 Customizing the Router Configuration (movie/index.js)

The controller is responsible for reading information from the model based on the information from the request, if necessary, and for rendering the view. The crucial point here is to distinguish whether the user passed an ID to edit a data record or to create a new one. As you can see in Listing 6.26, a data record is passed to the view in each case.

```
import { getAll, remove, get } from './model.js';
import { render } from './view.js';
import { render as form } from './form.js';

export async function listAction(request, response) {...}

export async function removeAction(request, response) {...}

export async function formAction(request, response) {
  let movie = { id: '', title: '', year: '' };

  if (request.params.id) {
    movie = await get(parseInt(request.params.id, 10));
  }

  const body = form(movie);
  response.send(body);
}
```

Listing 6.26 “formAction” in the Controller (movie/controller.js)

The implementation of this view is based on the implementation of the list. Listing 6.27 contains the source code.

```
export function render(movie) {
  return `
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8">
      <title>Movie list</title>
      <link rel="stylesheet" href=" /style.css" />
    </head>
    <body>
      <form action="/movie/save" method="post">
        <input type="hidden" id="id" name="id" value="${movie.id}" />
        <div>
          <label for="title">Titel:</label>
```

```

    <input type="text" id="title" name="title" value="${movie.title}" />
  </div>
  <div>
    <label for="id">Year:</label>
    <input type="text" id="year" name="year" value="${movie.year}" />
  </div>
  <div>
    <button type="submit">save</button>
  </div>
</form>
</body>
</html>
`;
};
```

Listing 6.27 Form View (movie/form.js)

The form uses the `action="/movie/save"` and `method="post"` attributes to ensure that the HTTP POST method data entered by the user is sent to the server. Based on the presence of an ID value, the server can distinguish whether the request is a creating or modifying operation. The values that are passed during editing are inserted into the individual form fields as `value` attributes via template substitutions. To be able to edit a record, you still need to implement the `get` method in the model, which you use to load the data record from the data source. In this application, it's sufficient to call the `find` method of the data array to load the data record based on its ID. Listing 6.28 shows the corresponding source code.

```
let data = [...];

export function getAll() {...}

export function get(id) {
  return Promise.resolve(data.find((movie) => movie.id === id));
}

export function remove(id) {...}
```

Listing 6.28 Extending the Model with the “get” Method (movie/model.js)

When reloading your application, you can reach the form either via the **New** or the **Edit** links in the list. The results are shown in Figure 6.9.

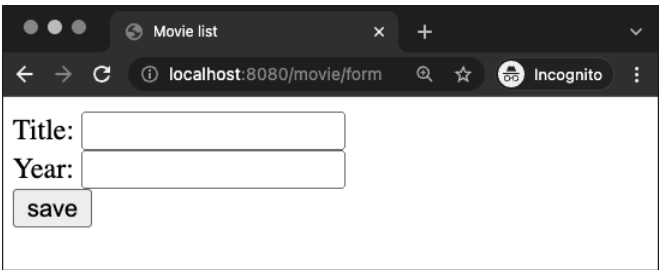


Figure 6.9 Form for Creating Data Records

6.8.1 Handling Form Input: Body Parser

To save your users' input, you must be able to access the form data. The most convenient way to do so is to use the body parser middleware. The body parser package has had a troubled past. Originally, middleware was an integral part of Express, similar to `static` middleware. However, with version 4, it was moved out into a package of its own, and with version 4.16, it was incorporated back into the core of Express. So, if you're using a recent version of Express, you don't need to install any additional packages to request body processing.

The body parser provides you with the two functions, `json` and `urlencoded` for JSON- and URL-encoded requests, respectively, which you can access directly via the `express` object. You can see the middleware integration in the `index.js` file of your application in Listing 6.29.

```
import express from 'express';
import morgan from 'morgan';
import { dirname } from 'path';
import { fileURLToPath } from 'url';
import { router as movieRouter } from './movie/index.js';

const app = express();

app.use(express.static(`${dirname(fileURLToPath(import.meta.url))}/public`));

app.use(morgan('common', { immediate: true }));

app.use(express.urlencoded({ extended: false }));

app.use('/movie', movieRouter);

app.get('/', (request, response) => response.redirect('/movie'));
```

```
app.listen(8080, () => {
  console.log('Server is listening to http://localhost:8080');
});
```

Listing 6.29 Integrating the Body Parser Middleware (index.js)

As already mentioned, the body parser middleware supports various parsers. If you're processing an ordinary HTML form, as in the example, you should use the `urlencoded` parser. If your frontend sends the information in JSON format instead, the JSON parser comes into play. Other parsers include the `raw` parser, which parses the body as a buffer, and the `text` parser, which interprets the body of the request as text. However, these two are only available as of Express version 4.17. You can also use the different parsers in parallel, for example, both the `urlencoded` and the JSON parser at the same time. After the inclusion, you must extend the router of your movie module to support the `/movie/save` URL path.

As you can see in Listing 6.30, the extension follows the scheme we've been using until now. The `saveAction` of the controller passes the information to the model. For better control, the controller extracts the required properties from the `request.body` property provided by the body parser. This property contains all the data of the form as object properties.

```
import { Router } from 'express';
import
  listAction,
  removeAction,
  formAction,
  saveAction,
} from './controller.js';

const router = Router();

router.get('/', listAction);
router.get('/delete/:id', removeAction);
router.get('/form/:id?', formAction);
router.post('/save', saveAction);

export { router };
```

Listing 6.30 Extending the Router with the Save Route (movie/index.js)

A separate view isn't needed in the controller because `saveAction` redirects to the list. Again, as with deleting data records, the `baseUrl` property of the `request` object comes into play. Listing 6.31 shows the customized controller.

```
import { getAll, remove, get, save } from './model.js';
import { render } from './view.js';
import { render as form } from './form.js';

export async function listAction(request, response) {...}
export async function removeAction(request, response) {...}
export async function formAction(request, response) {...}
```

```
export async function saveAction(request, response) {
  const movie = {
    id: request.body.id,
    title: request.body.title,
    year: request.body.year,
  };
  await save(movie);
  response.redirect(request.baseUrl);
}
```

Listing 6.31 "saveAction" of the Controller (movie/controller.js)

The controller doesn't know whether the current operation is a new creation or an update of a data record. This decision is left to the model. For this reason, the model code is also a bit more extensive at this point, as you can see in Listing 6.32.

```
let data = [...];

function getNextId() {
  return Math.max(...data.map((movie) => movie.id)) + 1;
}

function insert(movie) {
  movie.id = getNextId();
  data.push(movie);
}

function update(movie) {
  movie.id = parseInt(movie.id, 10);
  const index = data.findIndex((item) => item.id === movie.id);
  data[index] = movie;
}

export function getAll() {...}

export function get(id) {...}
```



```
export function remove(id) {...}

export function save(movie) {
  if (movie.id === '') {
    insert(movie);
  } else {
    update(movie);
  }
  return Promise.resolve();
}
```

Listing 6.32 Implementing the “save” Method in the Model (movie/model.js)

In the `save` function of the model, the `id` property of the transferred data is used to decide whether it’s a new or an existing data record. For new records, the `hidden` input field of the form isn’t filled or has an empty character string as its value. To keep the method clear and manageable, the `insert` and `update` functionality are swapped out into separate helper functions. The `insert` function uses the `getNextId` function, which searches for the next free ID in the data source. Normally, this task is assumed by the database; in our case, the highest ID of the data in the array is searched and incremented by one. Then the new ID is assigned to the record, and the information is pushed into the data array. When updating the data, you must first convert the ID of the record to a number because all information arrives from the client as character strings, and the ID is stored as a number for the calculation of the next higher ID in the data source. This change allows you to find the index of the affected record in the data source and adjust the array by overwriting the old record with the new information.

After restarting the process, you can now view, delete, edit, and create new movies in your database.

6.9 Express 5

The development of Express has lost some momentum. A clear sign of this is that there hasn’t been a major release in quite some time. In the summer of 2021, version 5 of the framework was still in alpha stage. Version 4 of Express was released in April 2014. Since then, the framework has received numerous minor updates. For web developers, however, this has a decisive advantage as well: Express is a very stable and thousand-times field-tested basis for your application, where you don’t have to fear that the API will change seriously.

For Express 5, the developers have announced that there will be no serious changes. Some methods that have proven to be impractical or misleading over time are removed. A classic example is the `send` method with which you could send a string or a number. If a number is passed, it’s sent to the user as a status. For example, an

Unauthorized message can be sent in a quick way. However, this has the disadvantage that you have no way to send a regular number to the client. With Express 5, you accomplish this with the `sendStatus` method.

Express 5, as long as it hasn’t yet been released, can be installed using the command `npm install express@5.0.0-alpha.8`.

6.10 HTTPS and HTTP/2

Because Express is based on the HTTP module of Node.js, the framework is quite flexible when it comes to exchanging the communication protocol.

6.10.1 HTTPS

Instead of HTTP, you can also use the secure HTTPS variant recommended for productive applications. For this, you don’t need to do anything more than pass the `app` object you created by calling the `express` function to the `createServer` method of the `https` module.

Listing 6.33 contains the necessary customizations to deliver your movie database with HTTPS. The example assumes that you’ve issued yourself a self-signed certificate and saved the files in the `cert` directory as in Chapter 5, Section 5.3. When you restart the server after these adjustments, you can reach your application at `https://localhost:8080`.

```
import { createServer } from 'https';
import { readFileSync } from 'fs';
import express from 'express';
import morgan from 'morgan';
import { dirname } from 'path';
import { fileURLToPath } from 'url';
import { router as movieRouter } from './movie/index.js';
```

```
const app = express();

app.use(express.static(`${dirname(fileURLToPath(import.meta.url))}/public`));

app.use(morgan('common', { immediate: true }));

app.use(express.urlencoded({ extended: false }));

app.use('/movie', movieRouter);

app.get('/', (request, response) => response.redirect('/movie'));
```

```
const options = {
  key: readFileSync('./cert/localhost.key'),
  cert: readFileSync('./cert/localhost.cert'),
};

createServer(options, app).listen(8080, () => {
  console.log('Server is listening to https://localhost:8080');
});
```

Listing 6.33 Express with HTTPS

6.10.2 HTTP/2

The integration of HTTP/2 works similar to HTTPS. However, the problem here is that Express version 4 isn't compatible with the HTTP/2 module of Node.js. Therefore, you have to switch to the `spdy` module at this point. Native support is planned for version 5.

You can install the `spdy` module using the `npm install spdy` command. The name of this module is somewhat misleading, as it not only supports SPDY but also HTTP/2. SPDY is a protocol developed by Google to replace HTTP in version 1. The HTTP/2 protocol picks up some concepts from the SPDY protocol.

The `spdy` module is API compatible with the `http` and `https` modules of Node.js, so it combines well with Express. With regard to the integration, you can proceed in a similar way as you did before with the HTTPS integration. As you can see in Listing 6.34, instead of importing the `https` module, you must import the `spdy` package and call the `spdy.createServer` function instead of the `createServer` function of the `https` module.

```
import spdy from 'spdy';
import { readFileSync } from 'fs';
import express from 'express';
import morgan from 'morgan';
import { dirname } from 'path';
import { fileURLToPath } from 'url';
import { router as movieRouter } from './movie/index.js';

const app = express();

app.use(express.static(`${dirname(fileURLToPath(import.meta.url))}/public`));

app.use(morgan('common', { immediate: true }));

app.use(express.urlencoded({ extended: false }));

app.use('/movie', movieRouter);
```

```
app.get('/', (request, response) => response.redirect('/movie'));

const options = {
  key: readFileSync('./cert/localhost.key'),
  cert: readFileSync('./cert/localhost.cert'),
};

spdy.createServer(options, app).listen(8080, () => {
  console.log('Server is listening to https://localhost:8080');
});
```

Listing 6.34 HTTP/2 in Express

You can verify that the switch to the HTTP/2 protocol worked by opening your browser's developer tools and making sure that, as in Figure 6.10, the **h2** protocol is used for loading each resource.

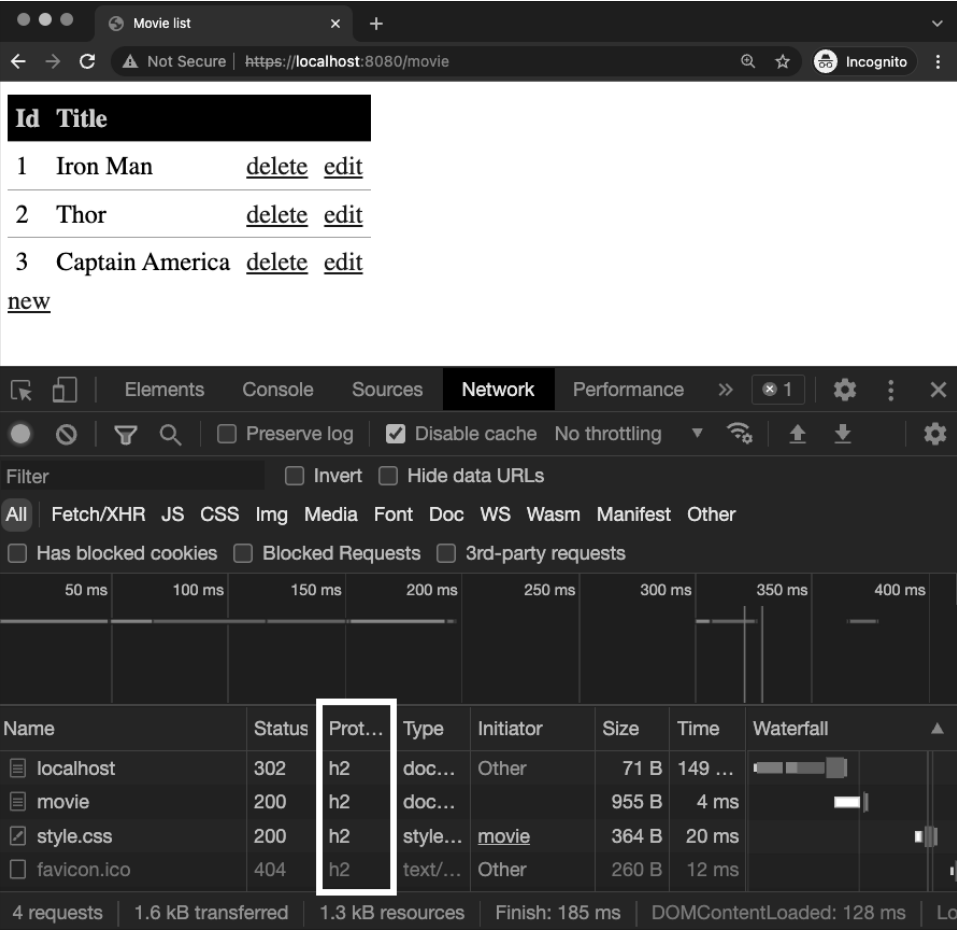


Figure 6.10 Delivery of Resources via the h2 Protocol

6.11 Summary

In this chapter, you learned about Express, the most widely used web application framework for Node.js. Unlike the `http` module of Node.js, it's much more convenient by either completely relieving you of numerous tasks or at least making them much easier. Express is a lightweight framework that is well suited for use in both small- and large-scale applications.

The Express framework provides a router that can be used to define combinations of `http` methods and URL paths and then bind callback functions to them. Routes in Express can have static and dynamic parts.

The middleware components provide you with a flexible plug-in system. A middleware refers to a function that is located between the incoming request and the outgoing response. You can use these functions to implement additional features, such as logging or request body processing, and use them to process, enrich, or log the request.

To extend your application, you can either use existing middleware components, such as Morgan or the body parser, or write your own components.

Due to its modular design, Express can be operated not only with HTTP but also with HTTPS and HTTP/2.

Chapter 15

Node on the Command Line

You cannot teach a man anything; you can only help him find it within himself.
— Galileo Galilei

In addition to typical web applications, Node.js can also be used to implement powerful command-line tools. This is possible because, with Node.js, you have an interface to your operating system that allows you to access not only the file system but also almost all aspects of your system. Another advantage of Node.js on the command line is that you can't tell whether an application implemented in Node.js is based on JavaScript. As you run your tool directly on the command line, there is no direct indication that Node.js is involved. The distribution of such tools is also quite simple thanks to Node Package Manager (npm). For open-source projects, you can use the infrastructure of the npm registry directly. You can distribute internal projects as files via npm or create a local repository. Chapter 25 describes how this works in detail.

The areas of use for command-line tools in Node.js are also very diverse. Starting from small utilities that support you in your daily development work up to large-scale applications, everything is possible. Most of the time, however, the commands are used in web development because JavaScript is one of the most commonly used languages in that area, and the relevant interfaces are already available. Thus, applications for handling CSS, HTML, and JavaScript are provided. Testing and analysis tools are also often written in Node.js. Another big area where Node.js is used is in the build process of web applications, where the goal is to prepare the source code so that it can be distributed to a server system and deployed.

15.1 Basic Principles

Before you start writing a command-line tool with Node.js, you need to know some basic principles about commands and how to use them. All commonly used operating systems have a command line where you can execute commands. This is true for macOS and Linux as well as for Windows. In Listing 15.1, you can see the execution of a typical shell command in a Unix environment.

```
$ ls -l /usr/local/lib/node_modules/
total 0
drwxrwxr-x 23 root   wheel   782 Dec 16 19:42 npm
drwxr-xr-x  8 nobody 41305271 272 Dec 23 02:01 nvm
```

Listing 15.1 Command-Line Command on a Unix Shell

A typical feature of a command-line tool is that it's called directly with no interpreter or server process required. Normally, you can also omit the file extension of the command. In addition, most commands are located in the system's search path, so they can be executed without an explicit path name. Only in the rarest cases is a command used on its own because you usually specify additional options and arguments to affect the execution. In the example in Listing 15.1, the `ls` command is used to create a listing of files in a directory. The `-l` option provides a more verbose output, and the `/usr/local/lib/node_modules` argument determines which directory you're interested in. This structure isn't arbitrary, but follows a convention that you should follow when creating a command-line tool with Node.js.

15.1.1 Structure

Node.js gives you a lot of leeway when building a command-line application. Although this is a great advantage for you because you're hardly restricted, it quickly turns into a disadvantage for the users of your application if every command on the command line has to be used differently. For this reason, you should make sure your commands always have the following structure: `<command> <options> <arguments>`. The individual components of a command line are as follows:

■ Command

The command designates the executable file of the tool. Normally, you can omit file name extensions such as `.exe` or `.bat`. If you haven't placed the file within the search path of your system, you must prefix the command with the absolute or relative path to the file. So, for frequently used commands, it's recommended to expand the system search path and copy the tool executable to a location that is included in the search path, or at least create a shortcut to it.

■ Options

The options of a command affect its behavior. This means you can use options to control what exactly your application should do. You've already seen a corresponding example in Listing 15.1 where the `-l` option made sure that more details were displayed. With regard to options, there is a convention that most applications follow. If you prefix an option only with `-`, the option should consist of only one letter. If the option requires a value, it's separated from the option by a space. Several options in the short notation can be combined. Thus, an `ls -l -a` becomes an `ls -la`. In the more verbose notation for options, you must use two `-` as prefix. The name of the option

in this case consists of a word, and the value is separated from the option by `=`. In this context, it isn't possible to group several options together as in the shorthand notation. An example of this notation is `grep --recursive --max-count=3 "node.js" *`.

■ Arguments

Arguments enable you to pass information to the command. For the directory listing in Listing 15.1, the argument consists of the name of the directory to be displayed. For example, the `grep` command accepts two arguments. If you run `grep --recursive "node.js" *`, the first argument is the character string `node.js` to search for, while `*` is a wildcard for all files and directories in the current directory.

As is so often the case in Node.js, you should be careful with the flexibility the platform gives you. For example, you can read the entire command line and determine the format of options and arguments yourself. This allows you to use a `%` rather than a `-` to indicate options. However, you'd better stick to the convention described earlier and thus allow the users of your application to use it as they are used to. By default, almost all commands support options such as `-h` or `--help` for a short help, so you can get a quick start with the features of the application.

15.1.2 Executability

To run a command on your system, you must meet some requirements. During the course of this chapter, you'll learn how to design your application. There are additionally some conditions that have to be fulfilled on the operating system side.

As mentioned earlier, the executable file of your application must be findable. If you install the application globally via the npm, you must make sure that the file is located in a directory that is in the search path of your system. If you want to install your application without the npm, you have to adjust your search path manually. On a Windows system, you can do this by extending the `PATH` variable in the system settings. You proceed in a similar way on a Unix-based system. Again, the environment variable is called `PATH` and contains a list of directories separated by colons.

Especially on Unix systems, you must set special permissions for an application to run on the command line. The Unix permission system provides for three types of permissions: read, write, and execute. You can set them for the owner of the file, the assigned group, and all other users of the system. For you to run the application, the executing user must have at least read and execute permissions on the file. If that isn't the case, you'll receive an error message informing you of the missing authorization. You can set the permission on the command line using the `chmod +rx index.js` command. This assumes that the starting point of your application is in the file named `index.js`. The `chmod` command in this case ensures that any user on the system can both read and execute the file. With this prior knowledge, you can now move on to creating your command-line application.

15.2 Structure of a Command-Line Application

As an example of a command-line application, we want to create an application that provides calculation tasks for the four basic arithmetic operations and checks the results you enter. In the following sections, you'll implement such an application step by step and see what options are available to you on the command line.

15.2.1 File and Directory Structure

Normally, a command-line application has at least two subdirectories:

- **lib**

The *lib* directory contains the actual application. Depending on the size of the tool, you can distribute the source code across several files and subdirectories. As an alternative to the name *lib*, you can also name this directory *src*. Both variants are quite commonly used.

- **bin**

The executable files are located in the *bin* directory of the application. If you follow this convention, it's easy for outsiders to get started with the application.

In addition to the directories and files of the application, there is also the package configuration in the form of the *package.json* file. According to the convention, you should store the *index.js* file, which is the entry point to your application, in the *lib* directory.

One of the features of a command-line application is that you can call the tool directly and don't need to use the `node` command first. For this purpose, you can make use of the *shebang* (`#!`) on Unix systems. This is a standardized character string that tells the system how to execute the script. To make sure the Math Trainer application runs on your system, you must create a file named *mathTrainer.js* in the *bin* directory. The contents of this file are shown in Listing 15.2.

```
#!/usr/bin/env node
```

```
import '../lib/index.js';
```

Listing 15.2 Math Trainer Executable File (*bin/mathTrainer.js*)

After you've made sure the execution permission is also set correctly on a Unix system, you can execute your application via the `bin/mathTrainer.js` command line in the root directory of your application.

15.2.2 Package Definition

One of the most important aspects is the *package.json* file of a project. It helps you to obtain an initial overview of a project. The file lists the name, description, and version

number as well as the dependencies to be installed. Furthermore, it references the entry point into the application. For the Math Trainer, you use the `npm init` command to create the *package.json* file.

The interactive wizard will ask you some questions, after which, you'll have an initial package configuration. This configuration file is primarily intended for normal Node.js applications and not for command-line tools. For this reason, you still need to make some adjustments. As a general best practice, you should set the `private` key to the value `true` so that you don't accidentally publish your application. As we're use the ECMAScript module system, you must define the value `module` as `type`. The `bin` object also represents a mapping from the command to the executable. If you've already created a file in the *bin* directory, as in the example, `npm init` will automatically create the mapping for you. You can delete the entries `main` for the entry point and `scripts` for various helper scripts for the time being. Listing 15.3 shows the *package.json* file for the Math Trainer application.

```
{
  "name": "math-trainer",
  "version": "1.0.0",
  "description": "A simple tool to train your math skills",
  "bin": {
    "math-trainer": "bin/mathTrainer.js"
  },
  "license": "ISC",
  "private": true,
  "type": "module"
}
```

Listing 15.3 “package.json” File for the Math Trainer Application

15.2.3 Math Trainer Application

Now that you've made the preparations for your application, it's time to implement the actual application logic. The user should be shown a certain number of tasks per basic calculating operation on the command line. They can choose between three levels of difficulty. At the first level, both operands are to be single-digit. At the second level, one of the two operands should be one-digit, the second two-digit, and, finally, at the third level, both should be two-digit. A special rule applies to division: only integer divisions should be possible, and to increase the difficulty a bit, you generate the two operands according to the rules mentioned before, multiply the first with the second operand, and use the result as the first operand.

First, you implement a helper function that generates a random integer operand for you. You store this function in the *lib/operands.js* file. Listing 15.4 shows the corresponding source code.

```
export default (digits) => Math.floor(Math.random() * 10 ** digits);
```

Listing 15.4 Helper Function to Create Random Operands (lib/operands.js)

The function exported as `default` expects a number that specifies how many digits the operand should have. With this number, you generate and return a corresponding integer via a combination of `Math.random`, `Math.floor`, and the exponentiation operator.

In the next step, you create a file named *task.js*, which you also save in the *lib* directory. This file contains the logic for generating the individual tasks. A task is represented by an object that has the properties `task`, `result`, and `input`. In *task*, you store the task as a character string. `result` contains the precalculated result of the task as a number, and `input` should finally contain the solution entered by the user and is first initialized with an empty string.

The *task.js* file contains two functions, as shown in Listing 15.5. The `createTask` method creates a new task object, and the `getOperands` method generates the two operators using the helper function from *operands.js*. Because you only need the `createTask` function outside the file, it also represents the default export of the file.

```
import createOperand from './operands.js';
```

```
export default function createTask(operation, level) {
  const [operand1, operand2] = getOperands(operation, level);
  const task = `${operand1} ${operation} ${operand2}`;
  const result = eval(task);

  return {
    task,
    result,
    input: '',
  };
}
```

```
function getOperands(operation, level) {
  let operands;
  switch (level) {
    case 1:
      operands = [createOperand(1), createOperand(1)];
      break;
    case 2:
      operands = [createOperand(1), createOperand(2)];
      if (createOperand(1) % 2 === 0) {
        operands.reverse();
      }
      break;
  }
}
```

```
case 3:
  operands = [createOperand(2), createOperand(2)];
  break;
}
if (operation === '/') {
  operands[0] = operands[0] * operands[1];
}
return operands;
}
```

Listing 15.5 Creating New Tasks (lib/task.js)

The `createTask` function creates two operands via the `getOperands` function and assigns them to the two variables `operand1` and `operand2` by means of a destructuring operation. With these two variables and the type of operation passed as the operator, the string representation of the operation is formed with a template string. You pass this string to the `eval` function to have the result calculated. The `eval` function executes a string as JavaScript source code. You should use this function only in exceptional cases, and then only if you have complete control over the character string being executed. This information forms the task object returned by the function.

The `getOperands` function receives as input the type of task and the difficulty level and then uses this information to generate the operands from the previously defined rules. At the second level of difficulty, the operands are randomly swapped using the `Array.prototype.reverse` method. For this purpose, you create an additional random number via the `createOperand` function. If it's an even number, it's swapped so that the two-digit operand can appear both in first and second place. At the end of the method, you must check if the operation is a division and adjust the first operand according to the task.

You can now test the functionality by creating an *index.js* file in the *lib* directory and integrating the application logic. Listing 15.6 shows the source code.

```
import createTask from './task.js';

const amount = 4;
const level = 2;
const operations = ['+', '-', '*', '/'];

operations.forEach((operation) => {
  for (let i = 0; i < amount; i++) {
    console.log(createTask(operation, level));
  }
});
```

Listing 15.6 Creating Tasks

The source code of the *index.js* file makes sure that four tasks per basic arithmetic operation are displayed to you when you run the application.

By implementing this file, you’ve created the final component for your command-line application, which means your application is theoretically functional. To test this, you can either install your application directly using the `npm install -g .` command, or you can use the `npm link` command. In this context, you enter the `npm link` command in the root directory of your application. `npm` takes care of everything else by ensuring that the application is installed globally. For this purpose, a symbolic link to the executable file is created in the global directory. In addition, the application directory is linked into the global *node_modules* directory. The advantage of `npm link` over an installation with `npm install` is that the link makes all changes to the application effective immediately, and you don’t have to reinstall the application. The command, `npm uninstall -g math-trainer` allows you to remove the link again when you’ve finished your development work.

Whether you choose to install or link, after running the command, you’ll be able to use Math Trainer system-wide. To do this, you enter the `math-trainer` command in the command line in any directory on your system. The result is shown in Listing 15.7.

```
$ math-trainer
{ task: '33 + 3', result: 36, input: '' }
{ task: '99 + 4', result: 103, input: '' }
{ task: '68 + 5', result: 73, input: '' }
{ task: '80 + 0', result: 80, input: '' }
{ task: '9 - 48', result: -39, input: '' }
{ task: '47 - 3', result: 44, input: '' }
{ task: '9 - 5', result: 4, input: '' }
{ task: '56 - 1', result: 55, input: '' }
{ task: '2 * 6', result: 12, input: '' }
{ task: '34 * 8', result: 272, input: '' }
{ task: '26 * 3', result: 78, input: '' }
{ task: '76 * 7', result: 532, input: '' }
{ task: '0 / 32', result: 0, input: '' }
{ task: '252 / 3', result: 84, input: '' }
{ task: '60 / 6', result: 10, input: '' }
{ task: '264 / 88', result: 3, input: '' }
```

Listing 15.7 Running Math Trainer

In the following sections, you’ll extend Math Trainer into a full-fledged application that a user can interact with.

15.3 Accessing Input and Output

In a web application, communication takes place over the network using a browser. The communication protocol is usually HTTP. For a command-line application, however, different rules apply when it comes to communication. There’s only one endpoint and not any number of them. Moreover, the user doesn’t connect to the application via the network, but works directly with the application through the command prompt. So, you have to keep some things in mind when it comes to input and output, especially if you aren’t just generating output but interacting with the user during the runtime of the command-line application, as in the Math Trainer example.

15.3.1 Output

The general rule for Unix applications is that if there’s nothing to report, the application won’t generate any output. If the processing was successful, it isn’t necessary to spend anything. Nevertheless, it’s good style to give the user direct feedback. This is either done automatically by the application or can be controlled by the user.

The simplest way to output information on the command line is the `console.log` method. Everything you pass to it is written directly to standard output. However, a Node.js process has two output channels: standard output and standard error output. You can address both output channels via the global `process` module. Listing 15.8 shows how you can access the output channels in write mode. The standard output channel is represented by the `process.stdout` object, while the standard error output channel is represented by the `process.stderr` object. Both objects are of the writable stream type and therefore implement the `write` method. Unlike `console.log`, which automatically inserts a line break, you have to take care of this yourself via the `write` method using the `\n` control character; otherwise, this method simply continues the current output line forever.

```
process.stdout.write('This is stdout\n'); // Output: This is stdout
process.stderr.write('This is stderr\n'); // Output: This is stderr
```

Listing 15.8 Accessing the Standard Output and Standard Error Output Channels

If you save the source code in a file named *output.js*, you can access the respective channel using the commands from Listing 15.9.

```
$ node output.js
This is stdout
This is stderr
$ node output.js 1> app.log
```

```
This is stderr
$ node output.js 2> err.log
This is stdout
```

Listing 15.9 Output in the Standard Output and Standard Error Output Channels

You won't notice any difference between the two output channels until you separate the two. You can use the `node output.js 2> err.log` command to redirect the standard error output to the *err.log* file so that only the values of the standard output are displayed. With `node output.js 1> app.log`, you write the standard output messages to the *app.log* file, and the error output appears on the console. In an application, this separation can be helpful to keep the output clear and still record all errors to handle them at a later time.

Similar to `console.log`, which writes to standard output, Node.js also provides `console.error`, which allows you to write directly to standard error output without having to go through the `process` module. In addition to these two admittedly most important features, the `console` object provides numerous other methods, such as `console.count`, which provides you with a counter, or `console.table`, which you can use to generate tabular output.

15.3.2 Input

An application not only consists of outputs but also responds to inputs to adjust the program flow accordingly. There are several possibilities for such an interaction with a command-line application. The easiest way is to use the standard input of the process. Like the output, the input is a data stream—in this case, a readable stream. With the `process.stdin` object, you have a reference to the standard input.

Listing 15.10 contains source code that you can use to accept data via standard input. If you save this source code in a file named *input.js*, you can start the example using the `node input.js` command. If you enter a character string via the keyboard and confirm the entry with the `[Enter]` key, the data is passed to the application and written to the standard output.

```
process.stdin.on('data', data => {
  console.log(data.toString());
});
```

Listing 15.10 Accessing the Standard Input

Not only can the standard input be operated using the keyboard, you can also redirect the output of other programs to your Node.js application. This output-input redirection, called *piping*, is achieved by connecting two commands with the pipe symbol (`|`). On a Unix system, for example, this works with the command chain `echo 'Hello world' | node input.js`. The Hello World string is written to standard output by the `echo`

command. This is forwarded by the pipe symbol to the standard input of the subsequent command. In this context, the string is parsed by the Node.js application, and appropriate output is generated.

Using the standard input to interact with the user turns out to be quite uncomfortable, especially with guided dialogs as you know them from `npm init`, for example. For this reason, Node.js has a second means of user interaction: the `readline` module.

15.3.3 User Interaction with the readline Module

Before you add the `readline` module to your application, you'll first learn how to use the module on the basis of a simple example. Listing 15.11 contains a code block that ensures the user is asked for their name and then greets them personally.

```
import { createInterface } from 'readline';

const rl = createInterface({
  input: process.stdin,
  output: process.stdout,
});

rl.question("What's your name? ", (name) => {
  console.log(`Hello ${name}!`);
  rl.close();
});
```

Listing 15.11 Personal Greeting to User

If you want to use the `readline` module, you have to include it first. In this case, you import directly the `createInterface` function from the `readline` module. To handle the input correctly, you use the `createInterface` function to generate an interface that you associate with the standard input and output of the current process. The interface of the `rl` object just created implements the `question` method, among other things. This method displays the specified string on the console and waits for input. When the input process is completed by pressing the `[Enter]` key, the callback function you passed as the second argument to the `question` method is called with the user's input. Once you've completed all interaction with the user, you must call the `rl.close` method to close the interface. If you don't do that, the application can't be closed properly because of resources that are still open.

The `readline` module is a good example of an asynchronous operation. To sequence multiple questions to the user, you either make another `question` call in the callback function of the first method call or use the asynchronous programming capabilities of Node.js. You'll learn more about this topic in the next chapter. At this point, only so much can be said: You can use a promise object to encapsulate the user's response.

Listing 15.12 contains an extension of Listing 15.11. In this case, the user is also asked for their place of residence.

```
import { createInterface } from 'readline';

const rl = createInterface({
  input: process.stdin,
  output: process.stdout,
});

function promisedQuestion(question) {
  return new Promise((resolve) => {
    rl.question(question, (answer) => resolve(answer));
  });
}

const user = {
  name: '',
  city: '',
};

user.name = await promisedQuestion('What's your name? ');
user.city = await promisedQuestion('Where do you live? ');

console.log(`Hello ${user.name} from ${user.city}`);

rl.close();
```

Listing 15.12 Asynchronous Combination of User Interactions

This special form of sequencing the `question` calls becomes necessary because the user is asked for the answers in an asynchronous way. This means that any second `question` call that immediately follows the first one is ignored by the process.

The core of the implementation in Listing 15.12 is the `promisedQuestion` function. It encapsulates the `question` method in a promise object and resolves it once the user has given their input. Using the top-level `await` feature of Node.js, you can then concatenate the two questions and output the result after answering the second question. At the end of the chain, you also execute the `rl.close` method to close the `readline` interface and thus terminate the application. If you run the sample code, you get an output like the one shown in Listing 15.13.

```
$ node readline.js
What's your name? Basti
```

```
Where do you live? Munich
Hello Basti from Munich
```

Listing 15.13 Output of the “`readline`” Example

With this information, you can now extend your Math Trainer implementation to ask the user for the results of the tasks. First, you need to slightly modify the `promisedQuestion` function, as you can see in Listing 15.14, so that you can use it here as well. Save this implementation in the `lib` directory under the name `promisedQuestion.js`.

```
export default function promisedQuestion(question, rl) {
  return new Promise((resolve) => {
    rl.question(question, (answer) => resolve(answer));
  });
}
```

Listing 15.14 Version of the “`promisedQuestion`” Function Adapted for Math Trainer (`lib/promisedQuestion.js`)

The adjustments to the `promisedQuestion` function are limited to passing a reference to the `readline` interface as the second parameter and exporting the function.

The further adjustments take place in the `index.js` file in the `lib` directory of the Math Trainer application. The updated version of the file is shown in Listing 15.15.

```
import { createInterface } from 'readline';
import createTask from './task.js';
import promisedQuestion from './promisedQuestion.js';
```

```
const amount = 4;
const level = 2;
const operations = ['+', '-', '*', '/'];
const tasks = [];
```

```
operations.forEach((operation) => {
  for (let i = 0; i < amount; i++) {
    tasks.push(createTask(operation, level));
  }
});
```

```
const rl = createInterface({
  input: process.stdin,
  output: process.stdout,
});
```

```
async function question(index) {
  const result = await promisedQuestion(`${tasks[index].task} = `, rl);
```



```

tasks[index].input = parseInt(result);
if (tasks[index].input === tasks[index].result) {
  console.log('Correct!');
} else {
  console.log('Wrong!');
}
if (++index < tasks.length) {
  question(index);
} else {
  rl.close();
}
}

```

```
question(0);
```

Listing 15.15 Integration of the “readline” Module

In the first step, you add the created task objects to the `tasks` array, which you’re going to use in the following steps. Then you generate the `readline` interface that you can pass to the `promisedQuestion` function. The `question` function represents the core of the application. It receives the index of the current task and calls itself until all tasks have been displayed to the user.

Once the user has entered the solution to a task via the command line, the promise of this question is resolved. Within the callback function, you save the input in the respective task object and then check whether the input was correct. If the task was solved successfully, the user will see the string `Correct!`. In the event of an error, the output should read `Wrong`. If there are more tasks to solve, you must call the `question` function again, or you can terminate the process using the `rl.close` method. If you’ve linked the application with `npm link`, you can test the implementation via the `math-trainer` command and get an output like the one shown in Listing 15.16.

```
$ math-trainer
```

```
4 + 68 = 72
```

```
Correct!
```

```
11 + 4 = 16
```

```
Wrong
```

```
2 + 24 =
```

Listing 15.16 Running Math Trainer

In some situations, you may need to limit the interaction with an application to pass options and arguments to enable the automation of an execution. In the following section, you’ll learn how to extend Math Trainer so that you can pass the difficulty level and the number of tasks via options.

15.3.4 Options and Arguments

At the start of this chapter, you saw how a command is structured, that options influence the behavior of a command, and that arguments provide additional information. The `argv` property of the `process` module allows you to access the command line of the application. It contains an array that stores the individual components of the command-line command used to invoke the current process. The first element of the `argv` array is the Node.js executable with the full path. The second element is the absolute path of the executed script, and all other elements map the options and arguments.

With this information at hand, you should now make sure that it’s possible to pass the `--level=<difficulty level>` and `--amount=<number of tasks>` options to the `math-trainer` command, which will affect the behavior of the application accordingly. For example, if you call Math Trainer via the `math-trainer --level=1 --amount=2` command, the structure of the `process.argv` array looks like the one shown in Listing 15.17.

```

[ '/usr/local/bin/node ',
  '/src/node/bin/math-trainer',
  '--level=1',
  '--amount=2' ]

```

Listing 15.17 Structure of the “process.argv” Array When Calling Math Trainer

To solve the task, you write a helper function called `getOptions`, which you swap out to a separate `getOptions.js` file in the `lib` directory. The source code of this file is shown in Listing 15.18.

```

export default function getOptions(levelDefault = 2, amountDefault = 4) {
  const level = getOptionValue(getOption('level'), levelDefault);
  const amount = getOptionValue(getOption('amount'), amountDefault);
  return {
    level,
    amount,
  };
}

```

```

function getOption(optionName) {
  return process.argv.find((element) => element.includes(optionName));
}

```

```

function getOptionValue(option, defaultValue) {
  if (option) {
    const [, value] = option.split('=');
    return parseInt(value, 10);
  }
}

```

```
    return defaultValue;
  }
}
```

Listing 15.18 Helper Function “getOptions” (lib/getOptions.js)

The helper function itself consists of the `getOptions` function, which is made available to the entire application as a default export. Within it, you extract both the difficulty level and the number of options from the command line. For both pieces of information, you define default values in the parameter list of the function in case no value is passed during the call. Because the operations for both pieces of information are the same, you can again swap them out to helper functions. The `getOption` function reads the passed option from the command-line array, while the `getOptionValue` function receives this information. Using a combination of the `split` method, which converts the option into an array, and the destructuring option, which assigns the value after the equal sign to the `value` variable, you extract the option value. Note that, at this point, the command line is interpreted as a character string, but your application works with integers. For this reason, you must convert the value to a number using the `parseInt` function. It’s also important to note that the functions are called first and only defined afterwards in this example. This works because JavaScript does something called *hoisting*. In this context, named functions such as `getOption` and `getOptionValue` are available in the entire scope, which, in this case, is within the file, no matter where you declare them. The situation is different for function expressions. Here you define a variable and assign a function object to it. Because this isn’t an atomic operation, the variable declaration is echoed, but the assignment isn’t, so you can’t use the function until after the assignment operation.

If the user hasn’t specified the option on the command line, the default value is used instead. In the `index.js` file, you now include the call of the helper function and can then use your application’s command-line options. The necessary adjustments are shown in Listing 15.19.

```
import { createInterface } from 'readline';
import createTask from './task.js';
import promisedQuestion from './promisedQuestion.js';
import getOptions from './getOptions.js';

const { amount, level } = getOptions();

const operations = ['+', '-', '*', '/'];
const tasks = [];

operations.forEach((operation) => {
  for (let i = 0; i < amount; i++) {
    tasks.push(createTask(operation, level));
  }
})
```

```
});

const rl = createInterface({
  input: process.stdin,
  output: process.stdout,
});

async function question(index) {
  const result = await promisedQuestion(`${tasks[index].task} = `, rl);
  tasks[index].input = parseInt(result);
  if (tasks[index].input === tasks[index].result) {
    console.log('Correct!');
  } else {
    console.log('Wrong');
  }
  if (++index < tasks.length) {
    question(index);
  } else {
    rl.close();
  }
}

question(0);
```

Listing 15.19 Integrating the Helper Function “getOptions” (lib/index.js)

Instead of assigning the two values in separate statements as before, you can use a destructuring statement at this point to assign the object returned by the `getOptions` function directly to the two constants. If you now start your application via the `math-trainer --level=1 --amount=1` command, you’ll see a total of four simple tasks.

15.4 Tools

Node.js has also established itself as a valuable tool on the command line. For this reason, you’ll find ready-made solutions for numerous problems in the area of command-line applications, which you can install as packages in your application. In the following sections, you’ll be introduced to three of these tools—Commander, chalk, and node-emoji—and integrate them into your Math Trainer.

15.4.1 Commander

As you’ve seen in the previous section, searching the command line for specific options involves a certain amount of work. The situation gets even more inconvenient at this

point if you also want to provide the shorthand notation of options instead of what we’ve done so far in the example. In this case, you normally don’t use equal signs as separators between the option and the value, so you also have to adjust the routine here. For parsing the command line, you can include Commander in your application. You can install the package via the `npm install commander` command. Because you’ve already swapped out the parsing of the command line to a separate file in the previous step, the adjustments for integrating Commander are limited to the *lib/getOptions.js* file. The updated version of this file is shown in Listing 15.20.

```
import program from 'commander';

export default (levelDefault = 2, amountDefault = 4) => {
  program
    .version('1.0.0')
    .option(
      '-l, --level <n>',
      'Difficulty level of tasks (1-3)',
      parseInt,
      levelDefault,
    )
    .option('-a, --amount <n>', 'Number of tasks', parseInt, ↻
      amountDefault)
    .parse(process.argv);

  const options = program.opts();

  return {
    level: options.level,
    amount: options.amount,
  };
};
```

Listing 15.20 Integrating Commander (lib/getOptions.js)

Due to the customization in Listing 15.20, the source code of your application has become simpler, and you also gained additional features. Thus, by default, Commander supports the `-V` and `--version` options to display the version of the application. In addition, when the application is invoked with the `-h` or `--help` option, a help block describing how to use the command is displayed.

After these changes, you no longer support only the long version of the options, but also a shortened version. If you call your application with the `-h` option, you’ll see an output like the one shown in Listing 15.21.

\$ math-trainer -h

```
Usage: math-trainer [options]

Options:

  -V, --version      output the version number
  -l, --level <n>    difficulty level of tasks (1-3) (default: 2)
  -a, --amount <n>   number of tasks (default: 4)
  -h, --help         output usage information
```

Listing 15.21 Display of the Math Trainer Help

Using the `option` method of the Commander package, you can define the individual options of your application. As the first argument, the method expects the name of the option. Here you can specify both the short and the long variant. If a value is to be passed to the application via the option, you can specify it afterwards. You have two different options for specifying the value: If you put the value in angle brackets, as in this example, it’s a mandatory value. If you want to define an optional option, you can use square brackets here. The second parameter of the `options` method represents the description of the option. This is displayed in the help menu. As a third argument, you can pass a function to manipulate the value. For the Math Trainer application, use the `parseInt` function to convert the passed value into a number. The last parameter allows you to pass a default value for the option. Commander then automatically inserts the string `default: <value>` into the option description.

For Commander to work, you must use the `parse` method to specify which data structure to evaluate. In most cases, this will be the `process.argv` array, but here you have the option to specify any array that follows the rules of `process.argv`.

All methods of the Commander object return the object itself, so that a fluent interface notation becomes possible, and you can directly concatenate the method calls.

You can obtain the values that were passed when the application was called by using the `opts` method. This method returns an object containing the individual options and their associated values as key-value pairs.

The Commander project site can be found at <https://github.com/tj/commander.js>. A lightweight alternative to Commander.js is available in the form of `minimist`. This module deals only with the correct parsing of command-line options. This project can be found at <https://github.com/substack/minimist>.

15.4.2 Chalk

A feature that is often underestimated is the formatting of the command line. It allows you to highlight important terms and thus guide the user on the command line. You

can apply colors and other formatting using control characters directly in `console.log`, for example. Listing 15.22 shows how this works.

```
console.log('\u001b[33m yellow');
console.log('\u001b[31m red');
console.log('\u001b[34m blue');
console.log('\u001b[0m');
```

Listing 15.22 Coloring the Console

The output of this example consists of the character strings `yellow`, `red`, and `blue`, each colored correspondingly. The last line resets the color of the console back to its original state. The string `\u001b[4m` allows you to underline the subsequent characters. Other features include italic, strikethrough, and bold font. You can also change the background color of the console. Admittedly, dealing with ANSI control characters in development isn't always convenient. Applying styles on the console is such a common problem that a module called `chalk` comprehensively solves this problem for you. It can be installed with the npm using the `npm install chalk` command. The code you implemented in Listing 15.22 via control characters can be implemented more elegantly with `chalk` using meaningful function names. The result shown in Listing 15.23 is the same as the one from the previous example.

```
import chalk from 'chalk';

console.log(chalk.yellow('yellow'));
console.log(chalk.red('red'));
console.log(chalk.blue('blue'));
```

Listing 15.23 Using Chalk

In your Math Trainer application, `chalk` enables you to format the output of the result in bold and green for success, and bold and red for failure. For this purpose, you must modify the `lib/index.js` file after installing the package, as shown in Listing 15.24.

```
import { createInterface } from 'readline';
import chalk from 'chalk';
import createTask from './task.js';
import promisedQuestion from './promisedQuestion.js';
import getOptions from './getOptions.js';

const { amount, level } = getOptions();

const operations = ['+', '-', '*', '/'];
const tasks = [];

operations.forEach((operation) => {
```

```
for (let i = 0; i < amount; i++) {
  tasks.push(createTask(operation, level));
}
});

const rl = createInterface({
  input: process.stdin,
  output: process.stdout,
});

async function question(index) {
  const result = await promisedQuestion(`${tasks[index].task} = `, rl);
  tasks[index].input = parseInt(result);
  if (tasks[index].input === tasks[index].result) {
    console.log(chalk.bold.green('Correct!'));
  } else {
    console.log(chalk.bold.red('Wrong'));
  }
  if (++index < tasks.length) {
    question(index);
  } else {
    rl.close();
  }
}

question(0);
```

Listing 15.24 Using Chalk in Math Trainer (lib/index.js)

As you can see in Listing 15.24, it's possible to apply several styles at the same time by concatenating the statements. Another convenient feature of `chalk` is that it also takes care of resetting the formatting to the console default style for you after the passed character string has been formatted.

The `chalk` project can be found on GitHub at <https://github.com/chalk/chalk>.

15.4.3 node-emoji

One of the most popular tools that uses emojis for console output is the package manager Yarn. Like `chalk`, emojis can be used to direct the user's attention to a particular output on the console, to make the console clearer because certain states can be expressed more quickly via an emoji than via text, and, finally, to liven up your application a bit by using the right emojis in the appropriate places. Because JavaScript supports the Unicode character set, it's possible to use Unicode emojis directly. An

alternative to this is to use the `node-emoji` package, which allows you to use the text representation of various emojis, making your code more readable.

Your implementation of the Math Trainer application is currently missing a summary of the results. To implement these in the `lib/summary.js` file, you can use the `node-emoji` package. In the first step, you install this package via the `npm install node-emoji` command. The source code in the `lib/summary.js` file is shown in Listing 15.25.

```
import emoji from 'node-emoji';

export default (tasks) => {
  const correctCount = tasks.reduce((correctCount, task) => {
    if (task.input === task.result) {
      correctCount++;
    }
    return correctCount;
  }, 0);
  const percent = (correctCount * 100) / tasks.length;
  if (percent === 100) {
    return emoji.emojify(
      `:trophy: Congratulations, you have solved all ${tasks.length} tasks correctly.`,
    );
  } else if (percent >= 50) {
    return emoji.emojify(
      `:sunglasses: Very good, you have correctly solved ${correctCount} out of
      ${tasks.length} tasks.`,
    );
  } else if (percent >= 1) {
    return emoji.emojify(
      `:cry: You have correctly solved ${correctCount} out of ${tasks.length} tasks,
      you can do better.`,
    );
  } else {
    return emoji.emojify(
      `:skull_and_crossbones: Your answers to all ${tasks.length} tasks are wrong.`,
    );
  }
};
```

Listing 15.25 Preparation of the Results Summary (`lib/summary.js`)

To display the result, you divide it into four categories: The user solved all tasks correctly, the user solved more than 50% correctly, the user solved less than 50% correctly, and the user didn't solve any task correctly. You can use the `emoji.emojify` method to generate a character string that contains an emoji. This method should be marked with colons. The `emoji.emojify(':trophy:')` call creates a character string containing the trophy emoji. In the last step you need to display the generated character string. You can do this by calling the helper function for summaries before you exit the process. Listing 15.26 contains the customized source code of the `lib/index.js` file.

```
import { createInterface } from 'readline';
import chalk from 'chalk';
import createTask from './task.js';
import promisedQuestion from './promisedQuestion.js';
import getOptions from './getOptions.js';
import summary from './summary.js';

const { amount, level } = getOptions();

const operations = ['+', '-', '*', '/'];
const tasks = [];

operations.forEach((operation) => {
  for (let i = 0; i < amount; i++) {
    tasks.push(createTask(operation, level));
  }
});

const rl = createInterface({
  input: process.stdin,
  output: process.stdout,
});

async function question(index) {
  const result = await promisedQuestion(`${tasks[index].task} = `, rl);
  tasks[index].input = parseInt(result);
  if (tasks[index].input === tasks[index].result) {
    console.log(chalk.bold.green('Correct!'));
  } else {
    console.log(chalk.bold.red('Wrong'));
  }
  if (++index < tasks.length) {
    question(index);
  } else {
    console.log(summary(tasks));
    rl.close();
  }
}
```



```

    }
  }

question(0);

```

Listing 15.26 Summary Display (lib/index.js)

In addition to formatting strings, the `node-emoji` package is also capable of resolving emojis in character strings or assigning them a random emoji. You can find the project site at <https://github.com/omnidan/node-emoji>.

15.5 Signals

On a Unix system, a *signal* is a message to a process. Such signals are often used to terminate a process. However, you can also just send some information to the process, for example, that the window size has changed. Most signals you send to a Node.js process cause an event to which you can bind a callback function to respond to the signal. For example, if a user presses the shortcut `Ctrl+C`, the `SIGINT` signal gets triggered. You can intercept this via `process.on('SIGINT', () => {})` and act accordingly. When integrating it into your Math Trainer application, you have to keep in mind that the `readline` interface intercepts the signals, so you can't respond to them directly. The solution to this problem is to register the event handler for the program termination not on the `process` object but on the `rl` object. Upon termination, the user should be shown a message that tells him how many tasks he has already solved until termination. For this purpose, you define another helper function and save it in the `lib/handleCancel.js` file. The source code of this file is shown in Listing 15.27.

```

export default (rl, tasks) => {
  rl.on('SIGINT', () => {
    const solvedCount = tasks.reduce((solvedCount, task) => {
      if (task.input !== '') {
        solvedCount++;
      }
    }, 0);
    console.log(
      '\nToo bad you want to leave, you only solved ${solvedCount} ↩️  

      of ${tasks.length} tasks.',
    );
    rl.close();
  });
};

```

Listing 15.27 Integrating a Signal Handler (lib/handleCancel.js)

You pass a reference to the `readline` interface and the task array to the helper function. First, you register a handler function for the `SIGINT` signal. In the callback function, use the `array-reduce` function to calculate how many tasks have been solved. Then you issue a message and end the process. This step is needed because a custom signal handler overrides the default, so it's no longer possible to exit the program via the `Ctrl+C` shortcut.

You still need to include the `handleCancel` function in the `lib/index.js` file now, so that you can intercept the signal correctly. Listing 15.28 shows the point at which you should integrate the function call.

```

import { createInterface } from 'readline';
import chalk from 'chalk';
import createTask from './task.js';
import promisedQuestion from './promisedQuestion.js';
import getOptions from './getOptions.js';
import summary from './summary.js';
import handleCancel from './handleCancel.js';

```

```
const { amount, level } = getOptions();
```

```
const operations = ['+', '-', '*', '/'];
const tasks = [];
```

```

operations.forEach((operation) => {
  for (let i = 0; i < amount; i++) {
    tasks.push(createTask(operation, level));
  }
});

```

```

const rl = createInterface({
  input: process.stdin,
  output: process.stdout,
});

```

```
handleCancel(rl, tasks);
```

```

async function question(index) {
  const result = await promisedQuestion(` ${tasks[index].task} = `, rl);
  tasks[index].input = parseInt(result);
  if (tasks[index].input === tasks[index].result) {
    console.log(chalk.bold.green('Correct!'));
  } else {
    console.log(chalk.bold.red('Wrong'));
  }
}

```

```
if (++index < tasks.length) {
  question(index);
} else {
  console.log(summary(tasks));
  rl.close();
}
}
```

```
question(0);
```

Listing 15.28 Integrating the Signal Handler (lib/index.js)

If you restart your application after making these adjustments, you can terminate the application at any time by pressing `[Ctrl]+[C]` and obtain a summary like the one shown in Figure 15.1.

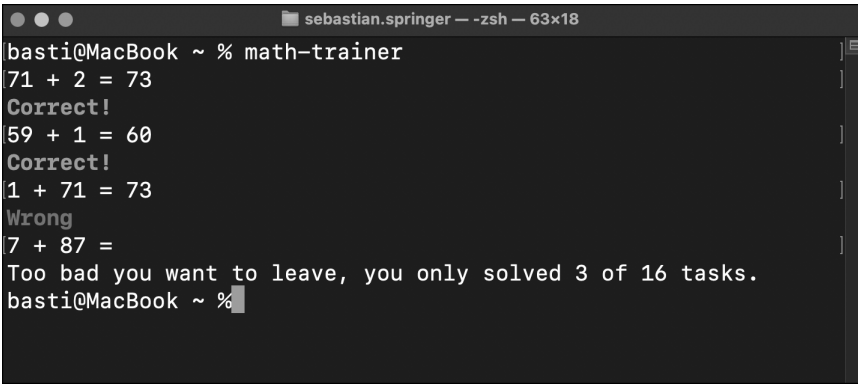


Figure 15.1 Terminating the Application

15.6 Exit Codes

Signals are means by which you can communicate with an application. Exit codes, on the other hand, work in exactly the opposite direction. In a way, an exit code is the return value of an application. On a Unix system, the `echo $?` command enables you to read the exit code of the last command on the command line. Usually, a Node.js application exits with exit code 0. This means that the application was terminated without any problem. An exit code with a value greater than 0 indicates an error.

Code	Name	Description
1	Uncaught Fatal Exception	An exception occurred that wasn't caught and caused the application to terminate.

Table 15.1 Exit Codes in Node.js

Code	Name	Description
3	Internal JavaScript Parse Error	The source code of Node.js itself caused a parse error.
4	Internal JavaScript Evaluation Failure	An error occurred while running Node.js.
5	Fatal Error	A fatal error has occurred in the V8 engine.
6	Nonfunctional Internal Exception Handler	An exception occurred and wasn't caught. The internal exception handler has been disabled.
7	Internal Exception Handler Runtime Failure	An exception occurred, wasn't caught, and the internal exception handler threw an exception itself.
9	Invalid Argument	An invalid option was passed during the call.
10	Internal JavaScript Runtime Failure	An exception occurred while bootstrapping Node.js.
12	Invalid Debug Argument	An invalid port was specified for the debugger.
>128	Signal Exit	If Node.js is terminated by a signal, the exit code 128 plus the value of the signal is set.

Table 15.1 Exit Codes in Node.js (Cont.)

Node.js automatically sets the correct exit code in most cases. However, you can also specify an exit code yourself. The `exit` method of the `process` module enables you to terminate the current process. This method accepts an integer as argument, which is used as an exit code.

15.7 Summary

Whenever you're faced with a problem you need to solve with a shell script, you can use Node.js. Especially when it comes to implementing solutions for automating tasks in the web environment, Node.js comes in handy. Many existing tools such as various CSS preprocessors, JavaScript optimizers, and HTML parsers show how processing web standards on the command line can work with Node.js.

You don't even have to look at such a Node.js command-line tool to know that it's a JavaScript application. Thus, such tools behave like the native commands. You can pass arguments and options to your Node.js application. For parsing the command line, you should use one of the available libraries such as Commander. As everywhere, the principle applies that you should first check the npm repository to see if there is already a solution to your problem before you start implementing it.

What a command-line application with Node.js is capable of is shown by a multitude of implementations that are used every day in web development, such as the build tool webpack.

Contents

Foreword	25
Preface	27
1 Basic Principles	31
1.1 The Story of Node.js	33
1.1.1 Origins	33
1.1.2 Birth of Node.js	34
1.1.3 Breakthrough of Node.js	34
1.1.4 Node.js Conquers Windows	35
1.1.5 io.js: The Fork of Node.js	36
1.1.6 Node.js Reunited	36
1.1.7 Deno: A New Star in the JavaScript Sky	36
1.1.8 OpenJS Foundation	37
1.2 Organization of Node.js	37
1.2.1 Technical Steering Committee	37
1.2.2 Collaborators	37
1.2.3 Community Committee	38
1.2.4 Work Groups	38
1.2.5 OpenJS Foundation	38
1.3 Versioning of Node.js	38
1.3.1 Long-Term Support Releases	39
1.4 Benefits of Node.js	40
1.5 Areas of Use for Node.js	40
1.6 The Core: V8 Engine	41
1.6.1 Memory Model	42
1.6.2 Accessing Properties	43
1.6.3 Machine Code Generation	45
1.6.4 Garbage Collection	46
1.7 Libraries around the Engine	47
1.7.1 Event Loop	48
1.7.2 Input and Output	50
1.7.3 libuv	50
1.7.4 Domain Name System	51
1.7.5 Crypto	52
1.7.6 Zlib	52

1.7.7	HTTP Parser	52
1.8	Summary	53
2	Installation	55
2.1	Installing Packages	56
2.1.1	Linux	57
2.1.2	Windows	60
2.1.3	macOS	63
2.2	Compiling and Installing	68
2.3	Node Version Manager	71
2.4	Node and Docker	71
2.5	Summary	72
3	Developing Your First Application	73
3.1	Interactive Mode	73
3.1.1	General Use	74
3.1.2	Other REPL Commands	75
3.1.3	Saving and Loading in the REPL	76
3.1.4	Context of the REPL	77
3.1.5	REPL History	77
3.1.6	REPL Mode	78
3.1.7	Searching in the REPL	78
3.1.8	Asynchronous Operations in the REPL	79
3.2	The First Application	79
3.2.1	Web Server in Node.js	80
3.2.2	Extending the Web Server	83
3.2.3	Creating an HTML Response	85
3.2.4	Generating Dynamic Responses	86
3.3	Debugging Node.js Applications	88
3.3.1	Navigating in the Debugger	90
3.3.2	Information in the Debugger	91
3.3.3	Breakpoints	93
3.3.4	Debugging with Chrome Developer Tools	96
3.3.5	Debugging in the Development Environment	97

3.4	nodemon Development Tool	98
3.5	Summary	99
4	Node.js Modules	101
4.1	Modular Structure	101
4.2	Core Modules	103
4.2.1	Stability	104
4.2.2	List of Core Modules	105
4.2.3	Loading Core Modules	108
4.2.4	Global Objects	111
4.3	JavaScript Module Systems	121
4.3.1	CommonJS	121
4.3.2	ECMAScript Modules	122
4.4	Creating and Using Your Own Modules	124
4.4.1	Modules in Node.js: CommonJS	125
4.4.2	Custom Node.js Modules	126
4.4.3	Modules in Node.js: ECMAScript	127
4.4.4	Exporting Different Types of Data	129
4.4.5	The modules Module	130
4.4.6	Module Loader	131
4.5	Summary	135
5	HTTP	137
5.1	Web Server	137
5.1.1	Server Object	137
5.1.2	Server Events	142
5.1.3	Request Object	145
5.1.4	Handling the Request Body (Update)	152
5.1.5	Delivering Static Content	157
5.1.6	File Upload	159
5.1.7	Fine-Tuning the Frontend	163
5.2	Node.js as HTTP Client	164
5.2.1	Requests with the http Module	164
5.2.2	The request Package	165
5.2.3	HTML Parser	167

5.3	Secure Communication with HTTPS	168
5.3.1	Creating Certificates	168
5.3.2	Using HTTPS in the Web Server	169
5.4	HTTP/2	170
5.4.1	HTTP/2 Server	170
5.4.2	HTTP/2 Client	173
5.5	Summary	175
6	Express	177
6.1	Structure	177
6.2	Installation	178
6.3	Basic Principles	179
6.3.1	Request	180
6.3.2	Response	180
6.4	Setup	181
6.4.1	Structure of an Application	182
6.5	Movie Database	185
6.5.1	Routing	186
6.5.2	Controller	189
6.5.3	Model	190
6.5.4	View	192
6.6	Middleware	193
6.6.1	Custom Middleware	194
6.6.2	Morgan: Logging Middleware for Express	195
6.6.3	Delivering Static Content	197
6.7	Extended Routing: Deleting Data Records	199
6.8	Creating and Editing Data Records: Body Parser	201
6.8.1	Handling Form Input: Body Parser	205
6.9	Express 5	208
6.10	HTTPS and HTTP/2	209
6.10.1	HTTPS	209
6.10.2	HTTP/2	210
6.11	Summary	212

7	Template Engines	213
7.1	Custom Template Engine	214
7.2	Template Engines in Practice: Pug	215
7.2.1	Installation	215
7.2.2	Pug and Express: Integration	216
7.2.3	Variables in Pug	219
7.2.4	Specific Features of Pug	221
7.2.5	Conditions and Loops	222
7.2.6	Extends and Includes	223
7.2.7	Mixins	226
7.2.8	Using Pug without Express	228
7.2.9	Compiling	228
7.3	Handlebars	229
7.3.1	Installation	230
7.3.2	Integration with Express	230
7.3.3	Conditions and Loops	232
7.3.4	Partials	234
7.3.5	Custom Helpers	236
7.3.6	Handlebars without Express	238
7.4	Summary	239
8	Connecting Databases	241
8.1	Node.js and Relational Databases	242
8.1.1	MySQL	242
8.1.2	SQLite	251
8.1.3	Object-Relational Mapping	257
8.2	Node.js and Nonrelational Databases	260
8.2.1	Redis	260
8.2.2	MongoDB	265
8.3	Summary	272

9	Authentication and Session Handling	273
9.1	Passport	273
9.2	Setup and Configuration	274
9.2.1	Installation	274
9.2.2	Configuration	274
9.2.3	Strategy Configuration	275
9.3	Logging In to the Application	277
9.3.1	Login Form	277
9.3.2	Securing Resources	280
9.3.3	Logging Out	281
9.3.4	Connecting to the Database	282
9.4	Accessing Resources	285
9.4.1	Access Restriction	285
9.4.2	Submitting Ratings	289
9.5	Summary	294
10	REST Server	295
10.1	Introduction to REST and Usage in Web Applications	295
10.2	Accessing the Application	296
10.2.1	Postman	296
10.2.2	cURL	297
10.3	Adaptations to the Application Structure	297
10.4	Read Requests	298
10.4.1	Reading All Data Records of a Resource	298
10.4.2	Accessing a Data Record	301
10.4.3	Error Handling	302
10.4.4	Sorting the List	304
10.4.5	Controlling the Output Format	307
10.5	Write Requests	309
10.5.1	POST: Creating New Data Records	309
10.5.2	PUT: Modifying Existing Data Records	312
10.5.3	DELETE: Deleting Data Records	314
10.6	Authentication via JWTs	316
10.6.1	Login	317
10.6.2	Safeguarding Resources	319

10.6.3	Accessing User Information in the Token	321
10.7	OpenAPI Specification: Documentation with Swagger	324
10.8	Validation	329
10.8.1	Installation and First Validation	330
10.8.2	Checking Requests with a Validation Schema	332
10.9	Summary	335
11	GraphQL	337
11.1	GraphQL Libraries	338
11.2	Integration with Express	339
11.3	GraphiQL	341
11.4	Reading Data via the Interface	342
11.4.1	Parameterizing Queries	345
11.5	Write Accesses to the GraphQL Interface	347
11.5.1	Creating New Data Records	347
11.5.2	Updating and Deleting Data Records	350
11.6	Authentication for the GraphQL Interface	353
11.7	Summary	355
12	Real-Time Web Applications	357
12.1	The Sample Application	358
12.2	Setup	358
12.3	WebSockets	364
12.3.1	The Server Side	366
12.3.2	The Client Side	367
12.3.3	User List	370
12.3.4	Logout	373
12.4	Socket.IO	377
12.4.1	Installation and Integration	378
12.4.2	Socket.IO API	379
12.5	Summary	383

13	Type-Safe Applications in Node.js	385
13.1	Type Systems for Node.js	386
13.1.1	Flow	386
13.1.2	TypeScript	390
13.2	Tools and Configuration	392
13.2.1	Configuring the TypeScript Compiler	393
13.2.2	Integration into the Development Environment	394
13.2.3	ESLint	395
13.2.4	ts-node	396
13.3	Basic Principles	398
13.3.1	Data Types	398
13.3.2	Functions	400
13.3.3	Modules	402
13.4	Classes	403
13.4.1	Methods	404
13.4.2	Access Modifiers	405
13.4.3	Inheritance	405
13.5	Interfaces	406
13.6	Type Aliases in TypeScript	408
13.7	Generics	409
13.8	TypeScript in Use in a Node.js Application	410
13.8.1	Type Definitions	410
13.8.2	Creating Custom Type Definitions	410
13.8.3	Sample Express Application	411
13.9	Summary	412
14	Web Applications with Nest	413
14.1	Installation and Getting Started with Nest	414
14.2	Nest Command-Line Interface	416
14.2.1	Commands for Operating and Running the Application	416
14.2.2	Creating Structures in the Application	418
14.3	Structure of the Application	419
14.3.1	Root Directory with the Configuration Files	419
14.3.2	src Directory: Core of the Application	420
14.3.3	Other Directories of the Application	420

14.4	Modules: Logical Units in the Source Code	421
14.4.1	Creating Modules	422
14.4.2	Module Decorator	423
14.5	Controllers: Endpoints of an Application	423
14.5.1	Creating a Controller	424
14.5.2	Implementing a Controller	424
14.5.3	Integrating and Checking the Controller	426
14.6	Providers: Business Logic of the Application	428
14.6.1	Creating and Including a Service	428
14.6.2	Implementing the Service	429
14.6.3	Integrating the Service via Nest's Dependency Injection	431
14.7	Accessing Databases	432
14.7.1	Setup and Installation	432
14.7.2	Accessing the Database	435
14.8	Documenting the Endpoints with OpenAPI	439
14.9	Authentication	442
14.9.1	Setup	442
14.9.2	Authentication Service	443
14.9.3	Login Controller: Endpoint for User Login	445
14.9.4	Protecting Routes	446
14.10	Outlook: Testing in Nest	449
14.11	Summary	451
15	Node on the Command Line	453
15.1	Basic Principles	453
15.1.1	Structure	454
15.1.2	Executability	455
15.2	Structure of a Command-Line Application	456
15.2.1	File and Directory Structure	456
15.2.2	Package Definition	456
15.2.3	Math Trainer Application	457
15.3	Accessing Input and Output	461
15.3.1	Output	461
15.3.2	Input	462
15.3.3	User Interaction with the readline Module	463
15.3.4	Options and Arguments	467

15.4 Tools	469
15.4.1 Commander	469
15.4.2 Chalk	471
15.4.3 node-emoji	473
15.5 Signals	476
15.6 Exit Codes	478
15.7 Summary	479
16 Asynchronous Programming	481
16.1 Basic Principles of Asynchronous Programming	481
16.1.1 The child_process Module	485
16.2 Running External Commands Asynchronously	486
16.2.1 The exec Method	487
16.2.2 The spawn Method	489
16.3 Creating Node.js Child Processes with fork Method	492
16.4 The cluster Module	496
16.4.1 Main Process	497
16.4.2 Worker Processes	501
16.5 Worker Threads	504
16.5.1 Shared Memory in the worker_threads Module	505
16.6 Promises in Node.js	507
16.6.1 Using util.promisify to Use Promises Where None Actually Exist	510
16.6.2 Concatenating Promises	511
16.6.3 Multiple Parallel Operations with Promise.all	512
16.6.4 Fastest Asynchronous Operation with Promise.race	513
16.6.5 Overview of the Promise Functions	514
16.7 Async Functions	514
16.7.1 Top-Level Await	516
16.8 Summary	517
17 RxJS	519
17.1 Basic Principles	520
17.1.1 Installation and Integration	521

17.1.2 Observable	521
17.1.3 Observer	522
17.1.4 Operator	523
17.1.5 Example of RxJS in Node.js	523
17.2 Operators	525
17.2.1 Creation Operators	527
17.2.2 Transformation Operators	529
17.2.3 Filtering Operators	532
17.2.4 Join Operators	534
17.2.5 Error Handling Operators	535
17.2.6 Utility Operators	537
17.2.7 Conditional Operators	538
17.2.8 Connection Operators	539
17.2.9 Conversion Operators	540
17.3 Subjects	540
17.4 Schedulers	542
17.5 Summary	543
18 Streams	545
18.1 Introduction	545
18.1.1 What Is a Stream?	545
18.1.2 Stream Usages	546
18.1.3 Available Streams	546
18.1.4 Stream Versions in Node.js	547
18.1.5 Streams Are EventEmitters	548
18.2 Readable Streams	548
18.2.1 Creating a Readable Stream	548
18.2.2 Readable Stream Interface	550
18.2.3 Events of a Readable Stream	550
18.2.4 Error Handling in Readable Streams	552
18.2.5 Methods	553
18.2.6 Piping	553
18.2.7 Readable Stream Modes	554
18.2.8 Switching to Flowing Mode	554
18.2.9 Switching to the Paused Mode	555
18.2.10 Custom Readable Streams	555
18.2.11 Example of a Readable Stream	556
18.2.12 Readable Shortcut	558

18.3 Writable Streams	559
18.3.1 Creating a Writable Stream	560
18.3.2 Events	560
18.3.3 Error Handling in Writable Streams	562
18.3.4 Methods	562
18.3.5 Buffering Write Operations	563
18.3.6 Flow Control	564
18.3.7 Custom Writable Streams	565
18.3.8 Writable Shortcut	566
18.4 Duplex Streams	566
18.4.1 Duplex Streams in Use	566
18.4.2 Custom Duplex Streams	567
18.4.3 Duplex Shortcut	567
18.5 Transform Streams	568
18.5.1 Custom Transform Streams	568
18.5.2 Transform Shortcut	569
18.6 Gulp	570
18.6.1 Installation	570
18.6.2 Example of a Build Process with Gulp	571
18.7 Summary	572
19 Working with Files	573
19.1 Synchronous and Asynchronous Functions	573
19.2 Existence of Files	575
19.3 Reading Files	576
19.3.1 Promise-Based API	581
19.4 Error Handling	582
19.5 Writing to Files	582
19.6 Directory Operations	586
19.7 Advanced Operations	589
19.7.1 The watch Method	591
19.7.2 Access Permissions	592
19.8 Summary	594

20 Socket Server	595
20.1 Unix Sockets	596
20.1.1 Accessing the Socket	598
20.1.2 Bidirectional Communication	600
20.2 Windows Pipes	602
20.3 TCP Sockets	603
20.3.1 Data Transfer	605
20.3.2 File Transfer	606
20.3.3 Flow Control	607
20.3.4 Duplex	609
20.3.5 Pipe	609
20.4 UDP Sockets	610
20.4.1 Basic Principles of a UDP Server	611
20.4.2 Example Illustrating the UDP Server	612
20.5 Summary	614
21 Package Manager	615
21.1 Most Common Operations	616
21.1.1 Searching Packages	616
21.1.2 Installing Packages	617
21.1.3 Viewing Installed Packages	622
21.1.4 Using Packages	623
21.1.5 Updating Packages	624
21.1.6 Removing Packages	625
21.1.7 Overview of the Most Important Commands	626
21.2 Advanced Operations	627
21.2.1 Structure of a Module	627
21.2.2 Creating Custom Packages	630
21.2.3 Node Package Manager Scripts	632
21.3 Tools for Node Package Manager	634
21.3.1 Node License Finder	634
21.3.2 Verdaccio	635
21.3.3 npm-check-updates	636
21.3.4 npx	637
21.4 Yarn	637
21.5 Summary	638

22	Quality Assurance	641
22.1	Style Guides	642
22.1.1	Airbnb Style Guide	642
22.2	Linters	643
22.2.1	ESLint	644
22.3	Prettier	648
22.3.1	Installation	649
22.3.2	Execution	649
22.4	Programming Mistake Detector: Copy/Paste Detector	649
22.4.1	Installation	650
22.4.2	Execution	651
22.5	Husky	652
22.6	Summary	653
23	Testing	655
23.1	Unit Testing	655
23.1.1	Directory Structure	656
23.1.2	Unit Tests and Node.js	656
23.1.3	Arrange, Act, Assert	657
23.2	Assertion Testing	658
23.2.1	Exceptions	661
23.2.2	Testing Promises	662
23.3	Jasmine	663
23.3.1	Installation	664
23.3.2	Configuration	664
23.3.3	Tests in Jasmine	665
23.3.4	Assertions	667
23.3.5	Spies	670
23.3.6	beforeEach and afterEach	671
23.4	Jest	671
23.4.1	Installation	671
23.4.2	First Test	672
23.5	Practical Example of Unit Tests with Jest	674
23.5.1	The Test	675
23.5.2	Implementation	676

23.5.3	Triangulation: Second Test	677
23.5.4	Optimizing the Implementation	678
23.6	Dealing with Dependencies: Mocking	679
23.7	Summary	681
24	Security	683
24.1	Filter Input and Escape Output	684
24.1.1	Filter Input	684
24.1.2	Blacklisting and Whitelisting	684
24.1.3	Escape Output	685
24.2	Protecting the Server	686
24.2.1	User Permissions	686
24.2.2	Problems Caused by the Single-Threaded Approach	688
24.2.3	Denial-of-Service Attacks	690
24.2.4	Regular Expressions	692
24.2.5	HTTP Header	693
24.2.6	Error Messages	695
24.2.7	SQL Injections	695
24.2.8	eval	699
24.2.9	Method Invocation	700
24.2.10	Overwriting Built-Ins	702
24.3	Node Package Manager Security	704
24.3.1	Permissions	704
24.3.2	Node Security Platform	705
24.3.3	Quality Aspect	705
24.3.4	Node Package Manager Scripts	706
24.4	Client Protection	707
24.4.1	Cross-Site Scripting	707
24.4.2	Cross-Site Request Forgery	709
24.5	Summary	711
25	Scalability and Deployment	713
25.1	Deployment	713
25.1.1	Simple Deployment	713
25.1.2	File Synchronization via rsync	715

25.1.3	Application as a Service	716
25.1.4	node_modules in Deployment	718
25.1.5	Installing Applications Using Node Package Manager	718
25.1.6	Installing Packages Locally	720
25.2	Tool Support	720
25.2.1	Grunt	721
25.2.2	Gulp	721
25.2.3	Node Package Manager	721
25.3	Scaling	721
25.3.1	Child Processes	722
25.3.2	Load Balancer	726
25.3.3	Node in the Cloud	728
25.4	pm2: Process Management	730
25.5	Docker	730
25.5.1	Dockerfile	731
25.5.2	Starting the Container	732
25.6	Summary	732
26	Performance	733
26.1	You Aren't Gonna Need It	733
26.2	CPU	734
26.2.1	CPU-Blocking Operations	734
26.2.2	Measuring the CPU Load	735
26.2.3	CPU Profiling with Chrome DevTools	736
26.2.4	Alternatives to the Profiler: console.time	738
26.2.5	Alternatives to the Profiler: Performance-Hooks Interface	739
26.3	Memory	741
26.3.1	Memory Leaks	742
26.3.2	Memory Analysis in DevTools	743
26.3.3	Node.js Memory Statistics	745
26.4	Network	747
26.5	Summary	751

27	Microservices with Node.js	753
27.1	Basic Principles	753
27.1.1	Monolithic Architecture	753
27.1.2	Microservice Architecture	755
27.2	Architecture	756
27.2.1	Communication between Individual Services	756
27.3	Infrastructure	758
27.3.1	Docker Compose	759
27.4	Asynchronous Microservice with RabbitMQ	759
27.4.1	Installation and Setup	760
27.4.2	Connecting to the RabbitMQ Server	762
27.4.3	Handling Incoming Messages	763
27.4.4	Database Connection	764
27.4.5	Docker Setup	765
27.5	API Gateway	768
27.5.1	Connecting the User Service	768
27.5.2	Asynchronous Communication with the User Service	770
27.5.3	Docker Setup of the API Gateway	774
27.5.4	Authentication	776
27.6	Synchronous Microservice with Express	780
27.6.1	Setup	781
27.6.2	Controller	782
27.6.3	Model Implementation	782
27.6.4	Docker Setup	784
27.6.5	Integration into the API Gateway	786
27.7	Summary	789
28	Deno	791
28.1	The Ten Things Ryan Dahl Regrets about Node.js	791
28.1.1	Promises	791
28.1.2	Security	792
28.1.3	The Generate Your Projects Build System	792
28.1.4	Package.json	792
28.1.5	Node_modules	792
28.1.6	Optional File Extension When Loading Modules	793
28.1.7	Index.js	793

28.1.8	What’s Going on Now with Node.js	793
28.2	Installing Deno	793
28.2.1	Deno Command-Line Interface	794
28.3	Execution	795
28.3.1	Running a TypeScript Application	796
28.4	Handling Files	796
28.4.1	The Task: Copying a File	797
28.4.2	Processing Command-Line Options	797
28.4.3	Reading Files	798
28.4.4	Permissions in Deno	800
28.4.5	readTextFile Function	801
28.4.6	Writing Files with Deno	801
28.5	Web Server with Deno	803
28.6	Module System	804
28.6.1	Loading External Modules into Deno	806
28.6.2	deno.land/x	807
28.6.3	Using Node Package Manager Packages	807
28.7	Summary	809
	The Author	811
	Index	813

Index

__dirname	112
__filename	112
.bom file	67
.eslint.json	395
.git	714
.gitignore	161, 622
.mjs	80, 110, 127
.msi package	60

A

Abstract class	556
Accessing properties	43, 291
Access restriction	285
Admin privileges	82
Aggregation	293
AJAX long polling	377
Analysis tool	453
Angular	295, 416
Annotation	423
Apache	137
API documentation	55, 441
Application	79
<i>server</i>	40
application/json	309
apt-get	59
Architecture	48
Argument	454
Array method	520
Arrow functions	49, 141
Assigning users	289
async_hooks	105
Async function	200
<i>async</i>	200, 514, 515, 581
<i>await</i>	200, 515
Asynchronicity	49, 119, 481
Asynchronous processing	484
Authentication	273
Authorizations	287
<i>header</i>	448
Automatic restart	98
Auxiliary program	453
Azure	60

B

Babel	571
Backtick	81

Backward compatibility	105
Base64	607
BDD	663
Bearer token	321
Best practice	129
Bidirectional communication	357
Bill of material	67
Binary data	112, 161
Binary package	60
Blocking operation	482
Block scope	81
Body parser	362, 769
Breaking changes	55, 104
Breakpoint	537
Browser	84, 137
BSD license	42
BSON format	265
Buffer	84, 112, 487, 579, 584, 601, 611
Bug fix	39, 55
Build	570
<i>process</i>	720

C

Caching	
<i>internal</i>	46
<i>removing a cache entry</i>	135
Callback	574, 582
<i>function</i>	483, 484
<i>hell</i>	514
C-Ares library	51
Case sensitivity	127
Certificate	209
Chain of commands	491
Chalk	471, 472
Change frequency	98
Changelog	104
Change of protocol	357
Character encoding	113
Cheerio	167, 192
child_process	485, 723
<i>application logic</i>	492
<i>callback function</i>	487
<i>child process</i>	493
<i>communication</i>	495
<i>configuration object</i>	492
<i>data stream</i>	489
<i>disconnect</i>	486

child_process (Cont.)	
exec	486, 487
execFile	486, 488
exec options	487
exit event	485
fork	492, 498
input file	489
kill method	485
message event	495
message handler	496
new Node.js process	494
output constraint	489
parent process	493
process.argv	494
send method	495
shell command	489
spawn	486, 489, 490
swapping out	492
Child process	485, 492, 722
cluster	726
locking	723
number	725
parallel process	724
performance	725
resource access	723
worker	724
worker process	723
chmod	455
Chrome	42
Chunk	140
Class	43
Client libraries	243
Cloud	728
Azure	729
Heroku	728
Heroku Toolbelt	728
platform as a service	728
Clustering	265, 496
cluster object	
cleanup	503
communication	497
communication link terminated	499
event	499
exit event	503
identification number	498
incoming connection	501
interface	501
isPrimary	498
isWorker	498
lifecycle	503
listening event	503
main process	497
cluster object (Cont.)	
message event	502
news	502
online event	501
process ID	501
properties	497
separate memory areas	497
setupPrimary	497
TCP port	504
terminated	501
unique identifier	501
Unix socket	504
worker objects	501
worker process	498, 501
Coding	84
Collaborator	37
Commander	469
Command line	58, 60, 73, 114, 453, 461
absolute path	467
affecting the behavior	454
ANSI control characters	472
application logic	457
argument	455, 467
argv property	467
asynchronicity	463
bin	456
bold	472
call	117
color	472
command	454, 467
command chain	462
createInterface	463
default style	473
default values	468
emoji	473
executability	455
execution permissions	455
exit code	478
exiting	477
formatting	472
help block	470
input	462
italic	472
lib	456
mandatory	471
message	476
notation, verbose	454
npm link	460
option	454, 467, 469, 471
output	461
package.json	456
package configuration	457

Command line (Cont.)	
package definition	456
parsing options	468
passing information	455
PATH variable	455
permission	455
pipe	462
question	463, 464
readline	463
requirements	455
return value	478
shorthand notation	454, 470
SIGINT signal	476
signal	476
strikethrough	472
structure	454, 456
tool	40, 453, 469
user interaction	463
version	470
Commit hook	648
CommonJS	108, 121
export	121
import	122
module system	80
Communication	595
protocol	241
Community committee	38
Compat Table	32
Compiler	45
Compiling	46, 68
Components	101, 125
Composite data type	385
Computed property	172
Computer architectures	56
Configuration file	58, 59
Console	113
console.error	462
const	81
Constant	81
Constructor	45
functions	43
Consul	756
Continuous integration	37
Contributor	37
cookie-session	362
Copy/Paste Detector	649
Core components	50
Core libraries	101
Core modules	77, 103, 105
loading	108
Correlation identifier	762
CouchDB	241, 719
CPD	
duplicate	649
execution	651
installation	650
Crockford, Douglas	643
CRUD	353
Crypto	52
CSS	157
preprocessor	53
cURL	296, 297, 300, 776
Cygwin	35, 60
D	
Daemon	
716	
Dahl, Ryan	33, 791
Database	102, 185, 241
abstraction layer	257
document-based	265
driver	242
foreign keys	242
non-relational	260
normalization	242
relationship	242
schema	242
Database driver	756
Data compression	52
Datagram	610
Data source	545
Data stream	149, 156, 196, 545, 573, 595
Data type	129, 385
composite	385
primitive	385
Debugging	40, 88
backtrace	91
breakpoint	93, 97
Chrome developer tools	96
commands	90
console.log	89
console messages	97
debugger	89
debugger statement	94
debug mode	91, 95
debug process	96
development environment	97
information	91
inspect	90
interactive debug mode	90
interactive shell	91
memory utilization	97
navigation	90
outputs	88

Debugging (Cont.)	
<i>remote targets</i>	96
<i>setBreakpoint</i>	93
<i>step by step</i>	93
<i>unwatch</i>	92
<i>variable scope</i>	97
<i>watch expressions</i>	97
<i>websocket</i>	90
Decorator	423
Deflate	52
Defragmented	47
Deleting a session	374
Deno	36, 791
<i>address-port combination</i>	803
<i>--allow-read</i>	800
<i>args</i>	797
<i>ArrayBuffer</i>	798
<i>asynchronous file system operation</i>	802
<i>body</i>	804
<i>cache</i>	806
<i>Chocolatey</i>	794
<i>CLI</i>	794
<i>command-line option</i>	797
<i>copying a file</i>	797
<i>deno.land/x</i>	807
<i>dependency</i>	806
<i>execution</i>	795
<i>export</i>	804
<i>external module</i>	806
<i>fibonacci</i>	807
<i>fileExists</i>	798
<i>file extension</i>	793
<i>file system</i>	796
<i>GYP</i>	792
<i>header field</i>	804
<i>homebrew</i>	794
<i>import</i>	804
<i>index.js</i>	793
<i>installation</i>	793
<i>JSPM</i>	808
<i>Linux</i>	793
<i>listen</i>	803
<i>Lodash</i>	808
<i>macOS</i>	793
<i>module system</i>	793, 804
<i>network permission</i>	804
<i>node_modules</i>	792
<i>Node.js core module</i>	807
<i>npm</i>	807
<i>package.json</i>	792
<i>permissions</i>	792, 799
<i>promise</i>	791
Deno (Cont.)	
<i>reading a file</i>	798
<i>readTextFile</i>	801
<i>sandbox concept</i>	800
<i>security</i>	792
<i>standard library</i>	806
<i>stat</i>	798
<i>synchronous file system operation</i>	802
<i>system resources</i>	792
<i>TextDecoder</i>	798
<i>TypedArrays</i>	801
<i>TypeScript</i>	796
<i>TypeScript compiler</i>	796
<i>Uint8Array</i>	799
<i>V8 engine</i>	795
<i>watch</i>	796
<i>web server</i>	803
<i>Windows</i>	793
<i>writing files</i>	801
Deployment	713
<i>.git directory</i>	714
<i>Artifactory</i>	719
<i>budget</i>	713
<i>by copying</i>	713
<i>dry-run</i>	715
<i>FTP server</i>	715
<i>local installation</i>	720
<i>Nexus</i>	719
<i>node_modules</i>	718
<i>npm</i>	718, 720
<i>npm proxy</i>	719
<i>npm script</i>	721
<i>open source</i>	719
<i>optimization</i>	713
<i>private repository</i>	719
<i>restart</i>	715, 716
<i>rsync</i>	715
<i>scp</i>	714
<i>server</i>	713
<i>Sinopia</i>	719
<i>synchronization</i>	714, 715
<i>tar archive</i>	720
<i>tool</i>	720
<i>Verdaccio</i>	719
Deprecated	104, 105
Destructuring	108, 346, 459
dgram	610
diagnostics_channel	105
Directory	44
<i>name</i>	112
<i>structure</i>	103, 131
Distribution	58

Django	177
DNS	51, 610
<i>resolution</i>	47
Docker	38, 71, 241, 244, 432, 730, 759, 765, 774, 784
<i>commands</i>	244
<i>container</i>	730
<i>Dockerfile</i>	731
<i>image</i>	72, 731
<i>start</i>	732
Docker Compose	759
<i>docker-compose.yml</i>	759
<i>Dockerfile</i>	759
<i>restart</i>	766
Docker Hub	243
DOM node	167
Duck typing	392
Duplicates	292
Dynamic web application	80
E	
EADDRINUSE	82, 600
ECMA-262	32
ECMAScript	32, 40
2015	141
<i>loader</i>	133
<i>version 6</i>	32
ECMAScript modules	122
<i>export</i>	122
<i>import</i>	123
ECMAScript module system	80, 108, 185, 516
<i>__dirname</i>	112
<i>__filename</i>	112
Embedded JavaScript	239
Encoding	140
Encryption	52, 357
Engine, optimized	42
Entry point	132
Environment variables	117, 132
Equality	150
Error	661
<i>handling</i>	303
<i>level</i>	584
ESLint	642, 644
eval	459
Event API	114
Event-driven architecture	48
Event-driven infrastructure	484
EventEmitter	548, 584
<i>emit</i>	548
<i>on</i>	548
Event loop	48, 113
EventTarget	114
Exception	523
Execution	80
Experimental	105
Express	177, 213, 413, 758
<i>__express</i>	217, 230
<i>alpha</i>	208
<i>baseUrl</i>	201
<i>basic principles</i>	179
<i>body parser</i>	180, 201, 205
<i>checkAuth</i>	364
<i>content-type</i>	180
<i>controller</i>	189
<i>cookie parser</i>	180
<i>custom middleware</i>	194
<i>data management</i>	190
<i>default views directory</i>	218
<i>dependencies</i>	188
<i>header</i>	180
<i>HTML, standards-compliant</i>	218
<i>HTTP/2</i>	209
<i>HTTPS</i>	209
<i>installation</i>	178
<i>json</i>	205
<i>log file</i>	196
<i>logging</i>	195
<i>middleware</i>	177, 193, 281, 319, 360
<i>model</i>	190
<i>modernization</i>	413
<i>module system</i>	187
<i>Morgan</i>	195
<i>next callback function</i>	195
<i>port</i>	179
<i>raw</i>	206
<i>render</i>	216
<i>request</i>	180
<i>request.body</i>	206
<i>request.session</i>	364
<i>response</i>	180
<i>route</i>	199, 360, 362, 363
<i>router</i>	178, 186, 202
<i>routing callback function</i>	189
<i>routing multipliers</i>	188
<i>routing patterns</i>	188
<i>routing variables</i>	200
<i>session</i>	362
<i>standard tasks</i>	195
<i>static content</i>	197
<i>static middleware</i>	197
<i>structure</i>	182
<i>text</i>	206

Express (Cont.)

urlencoded

version 5

view

view engine

express-handlebars

express-jwt

express-validator

External command

205

208

192

216

231

317, 353, 778

329

486

F

File extension

File name

extension

Files

File system

access

advanced operations

appendFile

asynchronicity

buffer

chmod

chown

close

constants

directory

encoding

error handling

error message

error object

exception

existence of files

file handle

flag

group

isDirectory

isFile

link

metadata

mode

nonexistent file

operations

path

path separator

permissions

promise

read access

readFile

search operation

stat

statistics

135, 454

112, 127

128, 454

573

50

575

589

585

573

579

593

593

585

576

586

580

579, 582

582

578

589

575

576, 582

578, 584

592

590

590

591

589

593

582

574

573

573

573, 592

573, 578, 581, 591

576

580

586

575, 590

589

File system (Cont.)

unwatch

user

watch

write access

writeFile

Filesystem Hierarchy Standard

File upload

Flash socket

Flow

flow.js

flowconfig

check

control

Facebook

591

592

591

582

585

57

159, 166

377

386

389

387

388

507

386

471

36

159

86, 152, 161

574

81

G

Garbage collector

mark-and-sweep

scavenge collector

GCC runtime library

Generalization

Git

GitHub

Global NPM packages

Global objects

Global scope

Global variables

Google Chrome

GraphQL

authentication

Boolean

buildSchema

creating

deleting

Express

express-graphql

Facebook

Float

graphqlHTTP

input type

Int

library

mandatory parameter

mutation

42, 46, 745

47

47

48

125

714

33, 104, 273

71

111

115

115

40

337, 341

353

343

338

347

350

339

339

337

343

340

348, 351

343

338

345

337, 347

GraphQL (Cont.)

parameters

query

reading

reference implementation

resolver

rootValue

schema

string

type system

updating

write

Group ID

Grunt

Gulp

build process

configuration

gulpfile.js

installation

NPM packages

plug-in

345

337, 339

342

338

337

343

337, 342

343

337, 343

352

347

118

570, 720

570, 720, 721

571

571

570

570

571

570

52

H

Handlebars

compile

conditions

curly brackets

Express

express-handlebars

extname

helper

installation

iteration

loops

partials

partial templates

placeholders

registering helpers

without Express

Hash value

Haskell

Hidden class

HMAC

Hoisting

Hostnames

HS256

HTML

form

page

parser

229

238

232

232

230

230

231

232, 236

230

232

232

234, 235

234

231

237

238

283

34

44

320

468

140

320

85, 144, 157

277

357

167

HTML (Cont.)

structure

HTML5

HTTP

body

cache

chunk

client

content type

createServer

default values

delivery

dynamic responses

forwarding

GET

header

HTTP header

message body

metadata

method

parameters

parser

POST

properties of the request object

protocol

queue

read access

request

request object

request package

resource

response header

response object

server

server events

server object

status code

structure of a request

submit form

text/html

trailer

version

write access

167

364

137, 357, 595

84, 145

149

156

164

84

139

165

143

86

151

86

84, 141, 145, 149

52

145

145

149, 165

86

52

86, 155

149

52, 137

140

165

164

84, 145

165

145

152

84

82, 497

142

137

84, 151

145

155

86

149

150

166

HTTP/2

client

connection setup

constants

server

session

socket connection

http module

36, 137, 170, 210

173

174

172

170

172

172

178

HTTPS 168

- certificates* 168
- key* 169
- web server* 169

https module 209

Husky 652

- Git hook* 652
- version control* 652

I

I/O operation 51

Image files 162

Images 157

import.meta.url 112

Importing directories 132

index.js 133

index.node 133

Input 50

--input-type 110

Installation 55, 56

Installer package 64

instanceof 385

Int32Array 506

Interactive mode 73

Interactive shell 73

Interchangeability 103

Interdependencies 378

Interface 101, 125, 368

Internal caching 46

Interpreter 454

Interval 614

io.js 36

IOCP 51

IP address 82, 139

IPv4 address 51, 82, 140

IPv6 address 82, 140, 611

ISO/IEC 16262 32

J

Jade 215

JaegerMonkey 42

Jasmine 663

- afterEach* 671
- assertion* 667
- beforeEach* 671
- command line* 665
- configuration* 664
- custom matcher* 669
- describe* 664
- error* 667

Jasmine (Cont.)

- execution* 665
- installation* 664
- it* 664
- jasmine.json* 664
- matcher* 668
- output* 666
- random execution* 666
- spec* 665
- spy* 670
- spyOn* 670
- test* 665
- test suite* 664
- toHaveBeenCalled* 670
- wrapper* 670

JavaScript 31

- engine* 34, 79
- pure* 41

Jest 671

- mocks* 680
- beforeEach* 673
- describe* 673
- execution* 672
- installation* 671
- jsdom* 671
- mock* 680
- test* 672

Joyent 35

jsconf.eu 35, 791

JScript 32

JSDoc 325

JSON 371, 576

- modules* 148

JSONP polling 377

jsontoxml 308

jsonwebtoken 317

JSON web tokens 298, 442

- sign* 318

Just-in-time compilation 42, 45

K

Key-value store 260

L

Layered architecture 177

Legacy 105

let 81

Let's Encrypt 168

libeio 50, 60

libev 49, 60

libevent 49

libevent2 49

Library 47

libuv 50, 60, 101

License information 61, 64

Link 70

Linter 643

- .eslintrc* 644
- error* 646
- ESLint* 643
- ESLint configuration* 644
- ESLint installation* 644
- ESLint version* 646
- formatting* 648
- integration* 648
- JSHint* 643
- JSLint* 643
- rules* 646
- style guide* 644
- warning* 646

Linux 55, 57

- binaries* 57

Linux Foundation 37, 38

LiveScript 31

Load balancer 497, 499, 726

- HAProxy* 726
- Least Connections* 728
- NGINX* 727
- proxy_pass* 727
- round robin* 727, 728

Load operations 135

Load process 132

Local repository 453

lodash 102, 617

Logger 582

Login 273, 360

- route* 279
- screen* 360

Long-lived memory area 47

Long-term support (LTS) 36, 39, 55, 56

Loose coupling 125

M

Machine code 45, 47

- generate* 45

Machine-to-machine communication 302

macOS 55, 63

Maintainability 494

Major release 39, 55

Make file 69

make install 70

MariaDB 243

Markup language 229

MD5 hash 283, 285

Memory 46

- area* 47
- defragmentation* 47
- model* 42
- range* 47
- section* 44

MessageChannel 115

MessageEvent 115

MessagePort 115

Message queue 757

Message system 115

Microservices 102, 753

- adapter* 753
- advantages* 755
- AMQP* 761
- amqp/rabbitmq* 763
- API gateway* 758, 768
- API gateway container* 774
- architecture* 753, 755, 756
- asynchronous communication* 757, 770
- authentication* 776
- authorization header* 779
- axios* 788
- bidirectional communication* 759
- communication* 755
- communication protocol* 761
- connectivity* 768
- consumer* 758
- container* 756
- controller* 782
- CRUD operations* 781
- cURL* 779, 789
- database* 782
- database connection* 764
- docker-compose.yml* 784
- docker-compose up* 767
- encryption* 768
- Express* 780
- fire and forget* 757
- HTTP* 770
- independence* 755
- infrastructure* 758
- interface* 755
- JSON web tokens* 776
- JWT authentication* 777
- login route* 777
- loose coupling* 755
- message handling* 763
- message object* 774

Microservices (Cont.)	
message queue	771
model	782
MongoDB container	765
MongoDB port	765
monolithic architecture	753
MVC structure	768
MySQL	781
public interface	769
RabbitMQ connection	762
RabbitMQ container	765
request/reply	771
security	768, 786
specialized module	754
synchronous communication	757
tight coupling	757
time-out	773
token	777, 779
Middleware	193, 353
minimist	471
Minor release	55
Mocha	31
Modularization	102, 125
Modular structure	101
module.exports	121
module.id	130
Module cache	133
Module loader	130, 131
Modules	81, 101
loading	81
own	124
modules module	130
Module system	79, 80, 115, 121, 146
absolute path	123
default export	123
export	148
import	115
module	115
named export	123
relative path	123
require	115
MongoDB	241, 265, 759, 764
abstraction layer	271
client	266, 267
collection	267
connection	266
creating data records	267
Docker	265
error object	269
find method	268
insert method	268
installation	266
MongoDB (Cont.)	
MongoClient	267
Mongoose	271
reading data records	268
removing data records	270
selector	270
unique ID	268
updating data records	269
Monolith	102
MSI installer	60
MSSQL	241
Multiclient chat	358
multipart/form-data	159
MVC	
controller	182
model	181
view	182
MySQL	241, 242, 432, 758
cascade	250
connection	245
container	243
createConnection	246
creating data records	247
database structure	244
delete	250
DELETE statement	250
Docker	243
escaping	247, 248
initialization	246, 433
insertId	248
installation	244
marking data records	251
password	244
placeholders	247
promise interface	248
promises	246
protocol	243
reading data records	246
removing data records	250
SELECT statements	250
updating records	248
mysql2	244
N	
Name conflicts	115
Named export	109
Namespace	44
Nest	177, 413
@nestjs	414
@nestjs/swagger	439
add	417

Nest (Cont.)	
ApiOkResponse decorator	440
ApiProperty decorator	440
app modules	415, 421
architecture	413
authentication	442
authentication service	443
auth module	445
body decorator	426
build process	416
CLI	414, 416
commands	417
config module	434
container	421
controller	423
controller decorator	425
createTestingModule	450
creating a controller	424
creating a module	422
creating a service	428
database	432
database access	435
decorators	423
delete	426
dependency	431
dependency injection	436, 449
dist	416
dist directory	420
DocumentBuilder	439
endpoint	424
entity	435
entity class	435
environment variable	434
factory	428
files	419
forFeature	436
forRoot	436
generate controller	424
generating a module	422
GET	426
guard	446
help	416
helper	428
HttpCode decorator	426
import	423
info	417
initialization process	415
injectable decorator	430
InjectRepository decorator	437
installation	414
JWT	443
JwtGuard	447
Nest (Cont.)	
JWT strategy	446
library	421
loading a service	431
middleware	413, 416
module	421
module decorator	423
monorepo	418
nest start	416
new command	414
npm start	415
OpenAPI	439
operating	416
package manager	414
param decorator	426
path variable	426
port	415
post	426
protecting route	446
provider	423, 428
put	426
registering a module	422
registering a service	428
repository	428
repository pattern	435
root directory	419
routing	413, 423
running	416
schematics	418
service	428
service implementation	429
service instance	431
src directory	420
status code	426
structure	413, 418, 419
Swagger	439
Swagger module	439
testing	449
testing module	450
TypeORM configuration	434
TypeORM module	434
TypeScript	413
TypeScript compiler	421
update	417
URL path	425
UseGuards decorator	447
user login	445
version	416
watch	417
net.Socket	601
Network communication	461
Nginx	137

Nitro	42
node_modules	161, 617
Node.js	34
areas of use	40
available system-wide	63
benefits	40
community	35
Node.js Foundation	36, 38
Node bindings	102
node-emoji	473
nodemon	98, 397
configuration file	99
debugging	99
log outputs	99
Node Packaged Module	615
Node Package Manager (npm)	34, 35, 60, 215, 414, 453, 615, 721
advanced operations	627
bundledDependencies	621
command-line tool	623, 626, 630, 637
commands	626
creating	630
deduped	622
dependency	619, 621, 622
devDependencies	621
directory hierarchy	619
engines	628
global	620
initialization	624
installation	615, 617, 633
keyword	616
license	630, 634
list	622
major version	628
minor version	628
ncu	636
nlf	635
node_modules	617
npx	637
operations	616
optionalDependencies	621
other sources	621
outdated	625
package.json	617, 621, 627
package-lock.json	618
patch level	628
peerDependencies	619, 621
private	632
--prod	622
production installation	628
proxy	636
publish	632, 633
Node Package Manager (npm) (Cont.)	
registry	616
repository	616, 630
restart	633
run	633
--save-dev	621
script	627, 632
search	616, 620
security risk	634
short forms	627
start	633
stop	633
structure	619, 627
system-wide installation	620
tar archive	617, 621, 631
test	633
tool	634
tree structure	622
uninstall	625, 633
unpublish	632
unpublish rules	632
update	624, 636
upload	631
usage	623
Verdaccio	635
version	624, 633
version control	621
version number	629
Node Security Platform	705
Node Version Manager	71
nodist	71
Nonblocking I/O	34, 41, 83
NoSQL	260
npm	
account	632
npm init	110, 457
npm init -y	137
npmjs	36
nvm	59
nvm-windows	71
O	
Object context	142
Object-oriented programming paradigm	43
Object-relational mapping	257
Observer pattern	519
Onion architecture	101
OpenAPI	324
OpenJS Foundation	37
Open source	241, 453
project	40, 56

OpenSSH	714
OpenSSL	52, 101, 168
Operating system	55, 69, 108, 603
Optimized engine	41, 42
Option	454
OracleDB	241
ORM	257, 756
ORM2	257
OSI	595
Output	50
channel	461
P	
package.json	110, 132, 185, 358, 706, 719
Package cache	59
package-lock.json	159
Package manager	56–58, 63, 615
Parallelizability	103
Parameter list	141
Parent module	131
Parent process	485
parseInt	200
Passport	273, 275, 277
accessing resources	285
authentication mechanism	273
browser cookie	275
configuration	274
connect-ensure-login	280
database	282
failure	277
installation	274
library	277
logging out	281
login	277
login form	277
logout method	282
logout route	281
middleware	273
modular authentication system	273
redirecting	279
securing resources	280
serialization	275
session	275
setup	274
strategy	273, 276
strategy configuration	275
Password	273, 363
plain text	285
Patch level	55
PATH variable	66
perf_hooks	116
Performance	116, 721, 733
abstraction	722
accurate measurement	741
activity monitor	735
advantage	43
allocation instrumentation on timeline	743
allocation sampling	743
analysis tool	733
arrayBuffers	746
asynchronicity	722
bandwidth	748
blocking operation	722, 734
bottleneck	733
cache	749
caching	722
child_process	734
code optimization	749
complexity	722
compression	749
computation	734
console.time	738
CPU	734, 736
data volume	733
debugger	738
DevTools	736
disabling the measurement	739
environment variable	739
execution times	738
external	746
flexible	733
garbage collector	741
getEntries	740
heap snapshot	743
heapTotal	746
heapUsed	746
hooks	36
htop	735
I/O operations	722
infinite loop	734
--inspect	736
--inspect-brk	736
lightweight	733
measure	740
measuring CPU	735
memory	741
memory address	741
memory analysis	743
memory consumption	743
memory leak	742
memory statistics	745
network	747
network statistics	748

Performance (Cont.)	
Network tab	748
observe	740
perf_hooks	739
performance-hooks	739
PerformanceObserver	739
pregeneration	722
process.memoryUsage	745
profiler	736
profiler recording	737
resident set size	746
retained size	743
rss	746
shallow size	743
speed	733
Task Manager	735
throttling	748
time to first byte	748
wait time	748
worker_threads	734
YAGNI	733
Persistence	241
PHP	177
Pipe	491
pkg file	64
pkg installer	68
pkgutil	67
Platform independence	55
Plug-in	273
pm2	730
cluster	730
pnpm	414
Polyfill	32
Port number	82
PostgreSQL	241
Postman	296, 300
Pre-commit hook	652
Prettier	648
.prettierrc	649
configuration	649
execution	649
formatting	648
installation	649
Prime number	495
Primitive data type	385
Process	116, 486
events	118
terminate	464
threads	486
Promise	119, 190, 507
catch	191
concatenation	511
Promise (Cont.)	
error case	508
error handler	512
fastest operation	513
finally	191
logical flow control	509
parallel operations	512
pending	508
Promise.all	512
Promise.race	513
reject	190
rejected	508
resolve	190
resolved	508
settled	508
statuses	508
success case	508
then	191
then method	509
Prototype	43, 182
Publish-subscribe	49, 757
Pug	215
base template	223
basic structure	225
blocks	223, 225
cache	228
compiling	228
conditions	222
dynamic elements	219
escaping	220
Express	216
extends	223
file extension	217
filtering	220
includes	223, 226
indentations	222
installation	215
interpolation	219
JavaScript expressions	219
library	227
loops	222
mixins	226
multiline blocks	221
parameter list	227
placeholders	228
program logic	222
rendering	216
reuse	225
specific features	221
structuring	221
syntax	217
template block	226

Pug (Cont.)	
variables	219
without express	228
Pull request	37
Python	177
Q	
Quality	
assurance	40, 641
metrics	641
static code analysis	641
tools	641
Query string	89
queueMicrotask	119
R	
RabbitMQ	
installation	757, 759
install	760
Rails	33
Random number	458
React	295
ReactiveX	519
Readability	494
Read requests	50
Real-time web application	357, 358, 377
Redis	241, 260, 722
backup copy	260
client	261
command set	265
connection	261
createClient	261
creating data records	261
Docker	260
editing data records	261
error object	262
events	261
installation	261
key	263
memory	260
overwritten	261
port	261
promise	262
reading data records	263
removing data records	264
ReferenceError	112
Relations	289
Relationship	435
Release schedule	38
REPL	73
.node_repl_history	77
REPL (Cont.)	
async	79
auto-complete	75
cancel	76
commands	75
context	75, 77
exit	76
global context	77
history	76, 77
location	78
magic	78
mode	78
multiline commands	75
saving and loading	76
sloppy	78
strict	78
Repository	58
require	108, 121, 134
resolve	135
Resolving NPM modules	131
REST	295, 337
204—no content	316
accept header	307
access	296
API documentation	327
authentication	316
checkSchema	333
content-type	309, 312
decoding tokens	321
delete	314
documentation	324
error handling	302
error message	318
Fielding, Roy	295
formats	297
HATEOAS	296
individual data records	301
JSON	295, 307
JSON web token	295, 317
links	305
login	317
output format	307
parameter validation	330
POST	309
PUT	309
query parameters	304
read requests	298
request.params	313
resources	296, 298
safeguarding resources	319
schema	332
sorting	304

REST (Cont.)		RxJS (Cont.)	
<i>stateless</i>	296, 298	<i>error handling</i>	522, 523, 535
<i>status code</i>	302	<i>event stream</i>	520
<i>structure</i>	297	<i>every</i>	539
<i>token</i>	320	<i>file system</i>	528
<i>unauthorized</i>	318	<i>filtering operator</i>	532
<i>updating data</i>	312	<i>first</i>	532
<i>validation</i>	329, 330	<i>flatMap</i>	531
<i>validation errors</i>	331	<i>fromEvent</i>	528
<i>validation schema</i>	332	<i>from operator</i>	527
<i>validator</i>	330	<i>hot observable</i>	523, 524
<i>write access</i>	309	<i>installation</i>	521
XML	307	<i>integration</i>	521
Restart	98	<i>iterable data structure</i>	527
Return	482	<i>join operator</i>	530, 534
<i>value</i>	74, 116, 141, 484, 507	<i>last</i>	532
Reuse	103, 125	<i>latestValueFrom operator</i>	540
Root directory	456	<i>logger</i>	524
root permission	57, 70	<i>map function</i>	529
Router	154	<i>marble diagram</i>	526
<i>implementation</i>	375	<i>mergeMap</i>	530
Routing	186	<i>merging observables</i>	534
RxJS	519, 545	Microsoft	519
<i>access to elements</i>	532	<i>multiple subscription</i>	540
<i>accumulator</i>	529	<i>next</i>	521
<i>aggregating</i>	529	<i>observable</i>	521
<i>array structure</i>	527	<i>observer</i>	519, 522
<i>asapScheduler</i>	543	<i>operator</i>	519, 523, 525
<i>asynchronicity</i>	519	<i>operator category</i>	526
<i>asyncScheduler</i>	543	<i>pipe method</i>	523
<i>AsyncSubject</i>	542	<i>promise</i>	527
<i>BehaviorSubject</i>	541	<i>queueScheduler</i>	543
<i>buffering</i>	529	<i>range</i>	529
<i>catchError</i>	535	<i>rejecting</i>	533
<i>cold observable</i>	523, 524, 539	<i>ReplaySubject</i>	541
<i>combineLatest</i>	534	<i>retry</i>	536
<i>complete</i>	521	<i>scan</i>	529
<i>complete method</i>	532	<i>scheduler</i>	542
<i>concatMap</i>	531	<i>skipUntil</i>	533
<i>conditional operators</i>	538	<i>startWith</i>	529
<i>condition for all data packages</i>	539	<i>subject</i>	540
<i>connection operators</i>	539	<i>subscribe</i>	522
<i>conversion operator</i>	540	<i>subscribe method</i>	522
<i>creation operator</i>	527	<i>subscription</i>	523
<i>data source</i>	519	<i>switchMap</i>	531
<i>debounceTime</i>	533	<i>take</i>	532
<i>debugging</i>	537	<i>takeUntil</i>	533
<i>decision tree</i>	526	<i>tap</i>	537
<i>defaultIfEmpty</i>	538	<i>termination</i>	522
<i>default value</i>	538	<i>time</i>	542
<i>documentation</i>	526	<i>time interval</i>	528
<i>error</i>	521	<i>timeout</i>	537

RxJS (Cont.)		Security (Cont.)	
<i>timeout operator</i>	538	<i>escaping</i>	697, 708
<i>timer observable</i>	534	<i>eval</i>	699
<i>toPromise</i>	540	<i>file system</i>	695
<i>transformation operator</i>	529	<i>filter input</i>	684
<i>trying again</i>	536	<i>function constructor</i>	700
<i>uniform interface</i>	519	<i>global functionality</i>	703
<i>unsubscribe</i>	523	<i>hash value</i>	695
<i>utility operator</i>	537	<i>header field</i>	693
<i>web server</i>	524	<i>hidePoweredBy</i>	695
S		<i>HTML injection</i>	707
Safety		<i>HTTP header</i>	693
<i>execution</i>	700	<i>I/O operations</i>	688
<i>Helmet</i>	694	<i>issue</i>	706
Scaling	40, 713, 721	<i>limiting child processes</i>	690
<i>cloud</i>	722	<i>link shortener</i>	708
<i>load balancing</i>	722	<i>logged-in user</i>	687
Schlueter, Isaac	34	<i>malicious code</i>	707
Scoping	741	<i>mapping table</i>	695
<i>block</i>	741	<i>method invocation</i>	700
<i>closure</i>	741	<i>modifying the database</i>	697
<i>function</i>	741	NoSQL	695
<i>global</i>	741	<i>npm</i>	704
<i>module</i>	741	<i>npm package</i>	703
Searching for modules	131	<i>npm script</i>	706
Search path	60, 70, 454	<i>NSP</i>	705
Secure Shell	714	<i>nvm</i>	704
Security	683	<i>origin header</i>	709
<i>administrator</i>	687, 704	<i>output data</i>	685
<i>attack scenario</i>	683	<i>overwriting</i>	702
<i>blacklist</i>	684	<i>OWASP</i>	693, 695
<i>blocking an IP address</i>	691	<i>permissions</i>	699, 704
<i>caching</i>	694	<i>preinstall</i>	707
<i>child process</i>	690	<i>query string</i>	685
<i>client</i>	707	<i>reading the database</i>	697
<i>configuration</i>	697	<i>ReDoS</i>	692
<i>contributor</i>	706	<i>regular expressions</i>	692
<i>CORS</i>	709	<i>repository</i>	705
<i>CPU-intensive operation</i>	688	<i>resource</i>	690
<i>cross-origin resource sharing</i>	709	<i>restricting the value range</i>	689
<i>cross-site request forgery</i>	709	<i>rimrafall</i>	704, 706
<i>cross-site scripting</i>	683, 686, 707	<i>same-origin policy</i>	709
<i>csrf</i>	710	<i>security gap</i>	701, 705
<i>database</i>	695	<i>security mechanisms</i>	686
<i>DDoS</i>	683	<i>security update</i>	39, 72
<i>denial of service</i>	690	<i>separate user</i>	687
<i>documentation</i>	705	<i>server</i>	686
<i>dynamic method call</i>	701	<i>session</i>	709
<i>error message</i>	695	<i>session hijacking</i>	683
<i>escape output</i>	684, 685	<i>software</i>	695
		<i>SQL injection</i>	695
		<i>standard function</i>	702

Security (Cont.)	
<i>statistics</i>	706
<i>system directory</i>	687
<i>token</i>	709
<i>user permissions</i>	686
<i>version number</i>	695
<i>web application</i>	686, 693, 707
<i>web store</i>	709
<i>whitelist</i>	684
<i>write permission</i>	687
Selectors	167
Self-signed certificate	170
semver	55
Sequelize	257
<i>configuration object</i>	258
<i>dialect</i>	258
<i>instance</i>	258
Sequential processing	484
Server object	82
Server process	499
Server request	510
Service	
<i>configuration script</i>	717
<i>crash</i>	716
<i>discovery</i>	756
<i>node-linux</i>	717
<i>node-mac</i>	717
<i>node-windows</i>	717
<i>systemd</i>	716
<i>Unix</i>	716
<i>Windows</i>	717
Session handling	273
SHA-256	320
Sharding	265
SharedArrayBuffer	505
Shared libraries	69
Shebang	456, 623
Shell	
<i>commands</i>	453
<i>interactive</i>	73
<i>script</i>	68, 492
<i>session</i>	58
Short-lived memory area	47
Shutdown	116
Side effect	133
SIGKILL	485
SIGTERM	117, 485
Single-page application	357
Single-threaded	33, 41, 688
<i>approach</i>	481
Socket	34, 595
<i>.sock file</i>	597
Socket (Cont.)	
<i>access</i>	598
<i>ACK package</i>	604
<i>addressing</i>	597, 602
<i>bidirectional communication</i>	600
<i>binary data</i>	606
<i>buffer</i>	608
<i>character encoding</i>	598
<i>checksum</i>	604
<i>client</i>	597
<i>close</i>	613
<i>coding</i>	606
<i>communication protocol</i>	595
<i>connection</i>	538, 597, 609
<i>data transfer</i>	605
<i>duplex</i>	609
<i>file system</i>	596
<i>file transfer</i>	606
<i>flexibility</i>	603
<i>flow control</i>	607
<i>fragmentation</i>	603
<i>handshake</i>	604
<i>multiple clients</i>	602
<i>network protocol</i>	605
<i>package size</i>	604, 614
<i>permission</i>	598
<i>pipe</i>	609
<i>port</i>	612
<i>security</i>	605
<i>SYN/ACK package</i>	604
<i>TCP</i>	603
<i>TCP client</i>	605
<i>TCP server</i>	605, 608
<i>UDP</i>	610
<i>UDP client</i>	611
<i>UDP server</i>	611
<i>Unix socket</i>	596, 607
<i>Windows pipe</i>	596, 602, 607
Socket.IO	377
<i>API</i>	379
<i>broadcast</i>	382
<i>client</i>	379
<i>connection</i>	380
<i>delivery</i>	379
<i>emit method</i>	381
<i>event</i>	381
<i>frontend</i>	379
<i>installation</i>	378
<i>integration</i>	378
<i>server</i>	382
Source code	70
Spdy	210

SQL	242
<i>injection</i>	247
SQLite	241, 251, 282
<i>all method</i>	256
<i>binary packages</i>	251
<i>connection</i>	252
<i>creating data records</i>	254
<i>database object</i>	252
<i>database structure</i>	251
<i>deleting data records</i>	256
<i>editing data records</i>	255
<i>escaping</i>	255
<i>file</i>	252
<i>GET method</i>	256
<i>INSERT query</i>	255
<i>installation</i>	251
<i>promise</i>	253
<i>reading data records</i>	253
<i>run method</i>	254
Stability	104
<i>index</i>	104, 122
Stable	105
Stack	486
Standard C library	48
Standard error output	461, 485
Standard input	462, 485, 588
Standard installation	66
Standard library	102
Standard output	461, 485, 491
Static content	157
Static page	277
Status code	304
stderr	116, 461
stdin	116, 462
stdout	116, 461
Stream	38, 545
<i>_read</i>	567
<i>_transform</i>	568
<i>_write</i>	567
<i>API</i>	607
<i>basic implementation</i>	565
<i>buffer</i>	563, 566
<i>buffer size</i>	549
<i>chain</i>	546, 548, 559, 568
<i>connection</i>	561
<i>cork</i>	564
<i>creating a readable</i>	548
<i>creating a writable stream</i>	560
<i>custom duplex stream</i>	567
<i>custom readable stream</i>	555
<i>custom transform stream</i>	568
<i>custom writable stream</i>	565
Stream (Cont.)	
<i>drain</i>	564
<i>duplex</i>	547, 566
<i>error event</i>	552
<i>error handling</i>	552, 562
<i>event handler</i>	550
<i>file system</i>	546, 548
<i>flow control</i>	564
<i>flowing mode</i>	551, 554
<i>framework implementation</i>	565
<i>highWaterMark</i>	549
<i>implementation</i>	548
<i>input</i>	546
<i>lifecycle</i>	550
<i>mode change</i>	554
<i>object</i>	486
<i>object mode</i>	558
<i>output</i>	546
<i>paused mode</i>	551, 554, 555
<i>permission</i>	562
<i>pipe</i>	553
<i>pipe method</i>	553, 569
<i>pull</i>	547, 554
<i>push</i>	547, 554
<i>readable</i>	547, 548, 607
<i>readable, sample stream</i>	556
<i>readable event</i>	549, 550
<i>readable method</i>	553
<i>readable stream API</i>	550
<i>readable stream mode</i>	554
<i>read method</i>	549
<i>shortcut</i>	558, 566, 567
<i>standard event</i>	550
<i>toString</i>	549
<i>transform</i>	547, 568
<i>transform shortcut</i>	569
<i>version</i>	547
<i>writable</i>	547, 559
<i>writable event</i>	560
<i>writable method</i>	562
<i>write permission</i>	562
Strict mode	78, 141, 699
Structure	
<i>large applications</i>	184
<i>medium-size application</i>	183
<i>small application</i>	182
<i>thematic separation</i>	185
Structuring	125
Style guide	642
<i>Airbnb</i>	642
<i>Google</i>	642
<i>JavaScript Standard</i>	642

Style guide (Cont.)	
<i>linter</i>	644
<i>standard</i>	642
Stylesheets	157
sudo	57, 620
Support period	39
Swagger	324
<i>swagger.json</i>	325
<i>Swagger specification</i>	327
<i>Swagger UI</i>	325, 439
Symbolic link	70, 460
Symfony	177
Syntax highlighting	312
System ports	82
System resources	485
System-wide	70
T	
Table	242
Target architecture	69
Target directory	62
TCP	595
<i>header</i>	604
<i>port</i>	139
<i>server</i>	497
TDD	674
Template	157
<i>language</i>	239
<i>strings</i>	81, 154, 215, 459
Template engine	185, 192, 213
<i>conditions</i>	215
<i>custom</i>	214
<i>loops</i>	215
<i>markers</i>	214
<i>parallelizability</i>	213
<i>performance</i>	215
<i>resources</i>	215
<i>reusability</i>	213
<i>security</i>	215
<i>separation of logic and markup</i>	213
<i>troubleshooting</i>	215
Test	655, 706
AAA	657
<i>act</i>	657
<i>arrange</i>	657
<i>assert</i>	656, 657
<i>assertion method</i>	659, 664
<i>assertion testing</i>	658
<i>assert module</i>	658
<i>async</i>	662
<i>automation</i>	655
Test (Cont.)	
<i>dependency</i>	674, 679
<i>directory structure</i>	656
<i>documentation</i>	655
<i>error</i>	659, 675
<i>exception</i>	661
<i>execution</i>	658
<i>fake it till you make it</i>	676
<i>feedback</i>	656
<i>frequent execution</i>	656
<i>fs mock</i>	679
<i>isolating</i>	655
<i>mocking</i>	679
<i>no manual intervention</i>	655
<i>only one test case</i>	655
<i>output</i>	658
<i>promise</i>	662
<i>quality</i>	656, 679
<i>refactoring</i>	678
<i>reject</i>	662
<i>relevant code</i>	656
<i>separate</i>	656
<i>side effects</i>	674
<i>strictEqual</i>	658
<i>strict mode</i>	660
<i>test directory</i>	656
<i>test first</i>	675
<i>throws</i>	661, 662
<i>triangulation</i>	677
<i>with source code</i>	656
Testability	103
TextDecoder	120
TextEncoder	120
this	141
Thread	486
<i>pool</i>	51
Tight cohesion	125
Tight coupling	753
Time-based features	113
Time-controlled calculation	358
Time-out error	83, 195
Timing functions	111, 113
TJ Fontaine	36
TJ Holowaychuk	177
Top-level await	79, 516
Top-level functions	34
Troubleshooting	89
Trustworthiness	683
try-catch	484, 659
type	127
<i>module</i>	110
Typecasting	660

Typed array	120, 506
typeof	385
TypeORM	432
<i>delete</i>	437
<i>find</i>	437
<i>findOne</i>	437
<i>save</i>	437
TypeScript	36, 390
<i>@types</i>	410
<i>@typescript-eslint</i>	396
<i>access modifier</i>	404, 405
<i>any</i>	399
<i>array</i>	398
<i>arrow function</i>	400
<i>async</i>	400
<i>Boolean</i>	398
<i>class</i>	403
<i>class Collection<T></i>	409
<i>collection</i>	409
<i>compiler</i>	391
<i>constructor</i>	404
<i>custom type definition</i>	410
<i>data type</i>	398
<i>declaration merging</i>	408
<i>default value</i>	401
<i>DefinitelyTyped</i>	410
<i>development environment</i>	394
<i>enum</i>	399
<i>ESLint</i>	395
<i>ESLint configuration</i>	395
<i>esModuleInterop</i>	394
<i>Express</i>	411
<i>extends keyword</i>	405
<i>forceConsistentCasingInFileNames</i>	394
<i>function</i>	400
<i>generics</i>	398, 409
<i>inheritance</i>	405
<i>initialization</i>	393
<i>installation</i>	391
<i>interface</i>	406
<i>method</i>	400, 404
<i>Microsoft</i>	390
<i>module</i>	393, 402
<i>module system</i>	391
<i>never</i>	399
<i>null</i>	399
<i>number</i>	398
<i>object</i>	400
<i>optional parameter</i>	401
<i>parameter</i>	400
<i>parameters list</i>	401
<i>placeholder type</i>	409
TypeScript (Cont.)	
<i>plug-in</i>	394
<i>polyfill</i>	391
<i>private</i>	405
<i>protected</i>	405
<i>public</i>	404, 405
<i>rest parameter</i>	401
<i>return value</i>	398
<i>shortcut notation</i>	404
<i>skipLibCheck</i>	394
<i>strict</i>	393
<i>string</i>	398
<i>target</i>	393, 403
<i>tools</i>	392
<i>tsconfig.json</i>	393, 402
<i>ts-node</i>	396
<i>tuple</i>	398
<i>type</i>	408
<i>type alias</i>	408
<i>type definition</i>	410
<i>type inference</i>	398
<i>undefined</i>	399
<i>union</i>	400
<i>unknown</i>	399
<i>void</i>	399
Type system	386
U	
Ubuntu	59
<i>desktops</i>	68
UDP	595
UglifyJS	571
Uint8Array	120
UnauthorizedError	354
UnauthorizedException	445
Unicode character set	473
Uninstalling	59, 63, 67
Unit test	655, 656
Unix	55
<i>philosophy</i>	101
<i>pipe</i>	545
URL	120, 145, 149
<i>parameter</i>	88
User ID	118
User name	273, 363
Users currently logged in	285
User table	282
UTF-8	84, 141
util.promisify	510
Utilities	72, 486

V

V8 engine 40, 41, 101, 104

V8 Inspector 96

Validity range 115

var 81

Variable 81

assignment 99

Version conflicts 132

Version control 714

system 619

Version information 63, 67, 117

Versioning 38, 55

Version number 59

View engine property 360

Visual Studio Code 97, 394

Vue 295

W

Waterline 257

Web application 753

dynamic 80

WebAssembly 121

Web development 453

Web server 35, 80, 137

HTTP response 80

WebSocket 102, 357, 364

API 357

client 367

connection 366

connection event 366

connection process 371

constructor 368

disconnecting 375

encrypted 365

logout 373

protocol 364

reference 372

send method 369

WebSocket (Cont.)

server 366

setup 376

status message 370

subprotocol 365

user 370

WebStorm 97

wget 57

WHATWG URL API 88

Windows 35, 50, 55, 60

Worker processes 497

Worker thread 36, 504

child thread 504

CPU 504

message event 505

postMessage 505

Work groups 38

Write operation 50

X

XML parser 616

Y

YAML 327

Yarn 386, 414, 473, 637

installation 637

node_modules 638

package.json 638

Plug'n'Play 638

pnpm 638

reliability 637

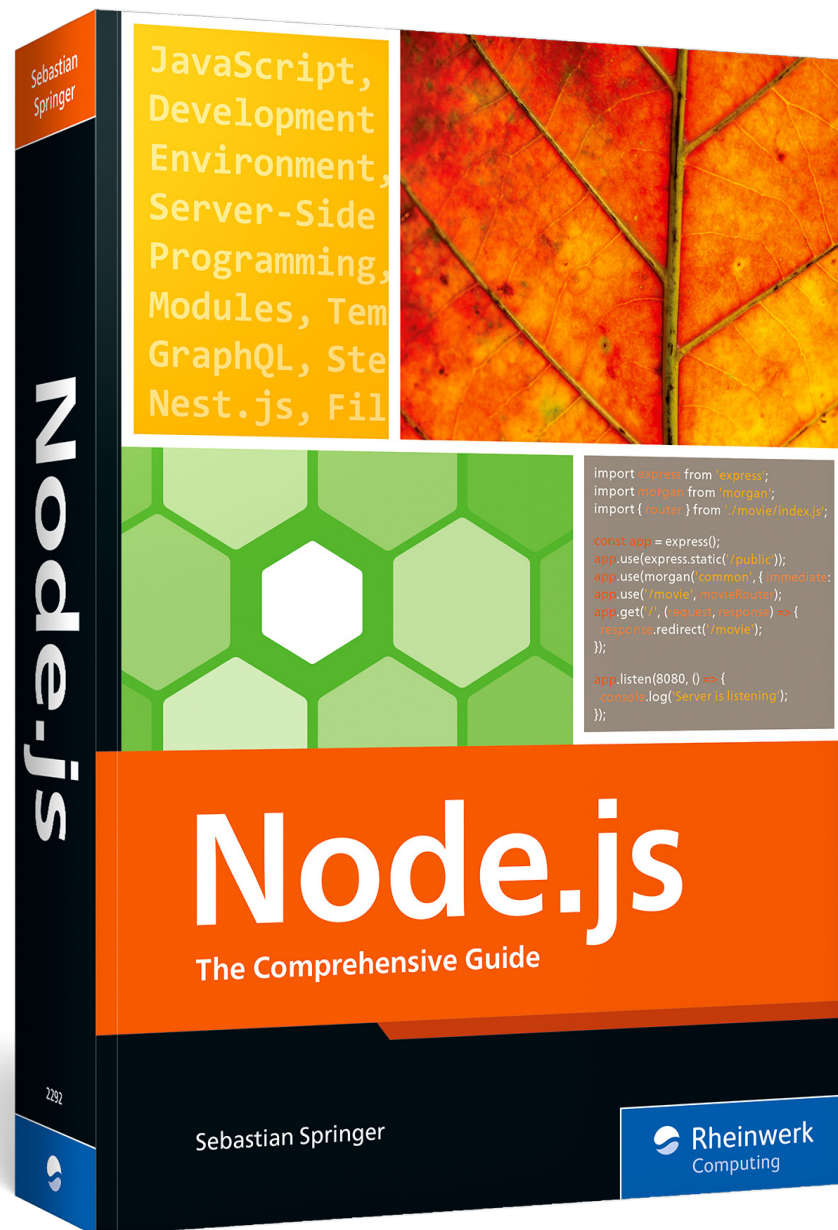
security 637

speed 637

yarn.lock 637, 638

Z

Zlib 52



Sebastian Springer is a JavaScript engineer at MaibornWolff. In addition to developing and designing both client-side and server-side JavaScript applications, he focuses on imparting knowledge. As a lecturer for JavaScript, a speaker at numerous conferences, and an author, he inspires enthusiasm for professional development with JavaScript. Sebastian was previously a team leader at Mayflower GmbH, one of the premier web development agencies in Germany. He was responsible for project and team management, architecture, and customer care for companies such as Nintendo Europe, Siemens, and others.

Sebastian Springer

Node.js: The Comprehensive Guide

834 pages, 2022, \$49.95
ISBN 978-1-4932-2292-6



www.rheinwerk-computing.com/5556

We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.