Building Production-ready Web Apps with Node.js

A Practitioner's Approach to produce Scalable, High-performant, and Flexible Web Components

GIREESH PUNATHIL

Building Production-ready Web Apps with Node.js

A Practitioner's Approach to produce Scalable, High-performant, and Flexible Web Components

GIREESH PUNATHIL

bpb

Building Production-ready Web Apps with Node.js

A Practitioner's Approach to Produce Scalable, High-performant, and Flexible Web Components

Gireesh Punathil

www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-91392-338

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

All those who took the time and had the patience to introduce me to the vast expanse of software engineering and its secret recipes of man-machine interactions, my known and unknown ancestors from whom I derived creativity with perseverance, my elder brother Sanal Kumar, who discovered my potential in computer science and channelized it, and my wife Savitha and children Parvathi and Adithya for their relentless support.

About the Author

Gireesh Punathil is a member of the Node.js Technical Steering Committee and an architect in IBM India Software Labs. In his 20-year career, he has been porting, developing, and debugging cloud platform components, web servers, language runtimes, virtual machines, and compilers. Gireesh graduated with an Electrical Engineering degree from Calicut University and is an IBM Master inventor who holds several patents on workload optimization and in related areas. He has also spoken at several international conferences, sharing his knowledge on Node.js and Java.

About the Reviewer

Dhruti Shah is a multi-skilled, tech savvy person with over 14 years of experience in education and training in Information Technology. She holds a Master of Computer Applications (MCA) degree from South Gujarat University and is a Microsoft Certified Training Specialist who has trained more than 2,000 candidates worldwide on over 10 technologies.

With a keen sense of technological advancement, she has self-acquired working knowledge of several technologies, including Java, .NET, PHP, Linux, SQL Server, MySQL, and Node.js. She has authored books on these technologies for educational institutions and also conducted webinars on technical subjects for international clients in Nigeria, Vietnam, and Trinidad.

She has been appreciated as a model representative of India for flawlessly managing two prestigious international collaboration projects to set up and upgrade the Centre of Excellence in Information Technology in Panama and Costa Rica, Latin America.

Acknowledgement

There is almost always the collective wisdom, with catalysts and motivators behind any achievement, although the author is the only visible part. I would like to thank my family for not only encouraging me for writing this book, but also sacrificing their personal time to give way for my work on this book, constantly reminding me of the timelines and rejuvenating and pushing me forward!

I am grateful to the Node.js community—the large number of known and unknown contributors, my fellow collaborators, my peers in the technical steering committee, and the OpenJS foundation—for providing me with a platform for experiencing the large and complex domain of web application, making a great shift from being largely involved in the lower-level stack components like language runtimes, virtual machines, compilers, and operating systems. I would like to thank them for encouraging me to write the book and reviewing it for the necessary legal aspects.

I am thankful to my employer organization IBM, which taught me the art of learning, interpreting, analyzing, synthesizing, and presenting software in its fundamental form.

Sincere thanks to Dhruti Shah for her thorough and valuable technical review of this book.

And finally, I am extremely grateful to BPB Publications, who contacted me and proposed this book with abundant confidence in me, without any experience of working with me on prior engagements but purely based on my open-source profile! This means a lot to me and is an exemplary embodiment of catalysts and motivators being behind every achievement.

Preface

It is an empirically proven fact in computer science that there is no single solution, language, platform, framework, or tool that can solve all computational problems. Instead, a suitable combination is carefully selected based on the characteristics of the workload. This means it is possible to find a language L1 that is better suited for a specific workload than another language L2, a framework F1 that works better for it than framework F2, and so on and so forth.

For example, the C programming language is used to develop device drivers, and JavaScript is used to develop interactive web pages. The reason why these languages are chosen is obvious—constructs that are often required for developing this code are readily available and in abundance in these languages.

On the other hand, when workloads evolve and mature (foreseen to stay so for a relatively long period), languages, tools, and frameworks are seen to be developed new (or evolve as well) to suit the needs of such workloads. The plethora of languages, tools, and frameworks that we see today is the result of such evolution—the continuous innovation in the software industry to improve the stated purpose of the business that runs these workloads.

Most modern workloads run on Cloud. A minimum common behavior exhibited by Cloud-hosted software is that they are accessed and consumed through a web interface. As a result, research has been performed to improve web workload efficiency. The most reasonable thing is to see how the programming platform can learn from the web characteristics and customize itself to improve developer productivity (the ability to write more code in less time) as well as machine productivity (the ability to run more code in less time).

Node.js is the successful result of such an innovation. By combining the constructs of JavaScript that enable asynchronous event-driven programming with the modern practices of developing in the open and packaging in the public, Node.js has pioneered the art of bringing

exponential performance improvement and developer productivity for highly concurrent web workloads.

By illustrating the step-by-step development of a production-grade application from scratch with Node.js, this book aims to unravel the power of Node.js. So, the most natural target audience of this book includes:

- Students who want to learn the essentials of web development with Node.js
- Developers who want to develop and specialize in various web components
- Architects who want to redefine, customize, and architect their application based on a specific use case by applying the learnings from Node.js story

This book covers various aspects of building a production-grade web application using Node.js by leveraging the essential components of a web server. We will not refer to or use any frameworks or modules for building our web application, so the book will discuss the internal capabilities of a web server in depth to illustrate the finest details of the features.

The book will cover several concepts, including event-driven architecture, asynchronous programming, website and web server, networking APIs, the basic building blocks of a web transaction (aka request-response cycle) at the frontend and at the backend, best practices for maintaining the enterprise-grade application, and the troubleshooting of common production issues.

This book is for anyone with basic programming fundamentals and some familiarity with client-server applications. While basic JavaScript knowledge is desired, prior experience of Node.js or backend web server is not required. We will cover the basics of backend server programming and a representative client's typical interactions with it.

As a result, this book takes the pragmatic approach of defining the basic building blocks of a web application, providing the example code in the backend, and showing the resulting view in the frontend. For each concept, we also put out questions, critically introspecting on the approach, presenting alternatives, discussing the tradeoffs, and ratifying the current design.

To introduce the concepts of enterprise enabling features like performance and security, the book builds a comparison model between a desktop application and a web application in the context of their exposure to various weaknesses. Identifying and addressing those weaknesses of the web application directly explains the relevance of the said features.

This book is divided into 11 chapters that cover the premise of highly concurrent workload, the elements of a web application, expanding each to a full-fledged component with code examples, and alternative design and implementation considerations. The details of the chapters are listed here:

Chapter 1 examines the basic premises of highly concurrent web workloads and the associated developer's considerations. It also covers the basics of event-driven architecture as the backbone of Node.js platform and the asynchronous programming style that it uses abundantly for event-driven architecture. This chapter also introduces the concept of concurrency versus parallelism, which plays an important role in the Node.js architecture.

Chapter 2 describes the essential pre-requisites for running the code we develop. We won't need a lot of preparation as we aim to build a web server from scratch by leveraging only the Node.js platform, but we need to make some subtle decisions to run our application as a stable web server that can handle real workload.

Chapter 3 discusses the fundamental considerations of a website developer. It examines the components that make up a web server and looks at a web server developer's main considerations to develop an efficient web server. It introduces concepts like website and client-server topology with various considerations around the application by virtue of its placement in the web backend.

Chapter 4 takes you through building a simple web server. The example code retrieves the time of the day on the locale with respect to the hosting server for any client that connects to it in any manner. It helps you dissect every section and line of this trivial program and try to make meaning out of it.

Chapter 5 digs deeper into the API abstractions that Node.js offers to implement web applications. It first differentiates the raw native networking protocol from a much higher-level and handy protocol. It also looks at

streaming APIs, as almost all server communications are stream-oriented. The chapter illustrates the two popular web server abstractions that represent the interaction: request and response. It also discusses in detail the configurations that affect the server's behavior and the server's life cycle control points and server events.

Chapter 6 is about the concepts of static and dynamic content serving that define the nature of the server. It also covers routes and endpoints that help the client and server categorize request types. It then introduces HTTP verbs (methods) and explains the most popular methods (GET and POST) in detail. Then, it illustrates how to forward a client request to another server using cookies and sessions for managing client sessions. Additionally, it touches upon common security issues and ways to address them. For all these components, it lays out the most common production issues and problem determination steps.

Chapter 7 examines other software components that our web server typically interacts with as part of serving our client. These could be services or modules developed as part of our application or third-party services that serve a specific purpose. It is essential to understand how and where the server interacts with these services to make our learning complete.

Chapter 8 illustrates a website's common requirements (frontend rendering) and how the pages and forms can implement some of those common features. We look at how a large amount of site data can be rendered for better consumption (pagination, search, and filtering), implementing authentication, authorization, and other common requirements.

Chapter 9 examines external factors to make our website robust and production grade. These factors include deployment topologies, scaling considerations, implementing RAS (Reliability, Availability and Serviceability), and monitoring and tracing. These ingredients ensure that the website is industrial-strength software that can be used at the enterprise level.

Chapter 10 explains some of the best practices that we need to follow from the runtime (virtual machine) perspective to make the server efficient. Some of these are generic to any runtimes, while a few are specific to Node.js.

Chapter 11 shares insight on the most common problem symptoms for a Node.js-based web server and provides a set of methods and best practices to be followed for each symptom to troubleshoot and resolve the issue comprehensively. The chapter covers the troubleshooting steps for crash, hang, spins, memory issues, performance, and exceptions.

Downloading the code bundle and coloured images:

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/d700d3

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at **www.bpbonline.com** and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit **www.bpbonline.com** and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Building-Production-ready-Web-Apps-with-Node.js**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at **https://github.com/bpbpublications**. Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**.

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential

readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

1. Getting Started with the Fundamentals
- Structure
- Objective
- Introduction to event-driven architecture
  - *Calculator example*
  - *Number sorting example*
  - *Echo example*
  - *Web page access example*
  - *Web server example*
  - *Windows event loop*
  - *Messaging example*
- Introduction to asynchronous programming
  - *Program interrupt example*
  - *Multimedia example*
  - *JavaScript web page example*
  - *Operating system scheduler example*
- Introduction to functional programming
  - *Function passed as a value*
  - *Function returned as a result*
  - *Function as assignment subject*
  - *Closure functions*
  - *Closure context*
- Concurrency
  - *Cooking example*
  - *Progress bar example*
  - *Garbage collection*
- Parallelism
  - *Distributed word counting example*
- Concurrency versus parallelism
  - *Similarity*
  - *Differences*
- Concurrency and scalability
  - *Thread pooling*

        Horizontal scaling
    Introduction to Node.js
        Latency in computer systems
        CPU bound and I/O bound operations
        Characteristics of web workload
        Bringing the context of scalability
        Bringing back the context of asynchronous programming
        Bring back the context of event-driven architecture
        Bring back the context of functional programming
    Core Node.js features
        Structure
        APIs
        Streams
        Small core philosophy
        npm
    Conclusion
    Points to remember

2. Setting Up the Environment
    Structure
    Objective
    Platform selection
    Node.js version selection
    Dependencies selection
    Resource requirements
    Code editor selection
    Conclusion

3. Introduction to Web Server
    Structure
    Objective
    Introduction to web server
        Core components
            Client
            Web layer
            Middleware
            Business logic layer

The microservice layer
Concerns and considerations of web servers
Comparison of a desktop and a web server
Similarities
Differences
Web server considerations: Architecture
Microservice architecture
Web server considerations: Performance
Performance overheads
Best practices
Web server considerations: Security
Security threat types
Best practices
Web server considerations: Reliability
Examples of reliability issues
Best practices
Web server considerations: Extensibility
Examples of extensibility
Best practices
Web server considerations: Maintainability
Importance of maintainability
Best practices
Web server considerations: Serviceability
Best practices
Web server considerations: Observability
Importance of observability
Best practices
Conclusion

4. Our First Program: Time of the Day Server
Structure
Objective
A ‘hello world!’ server
Running the program
Accessing the server through the browser
A Time of the Day Server
Program sections

Require statement
The server loop
The date
The server response
The server listen
Conclusion
Exercises
Problem #1
Problem #2

5. Common Networking Interfaces of Node.js
Structure
Objective
Network programming
Why a protocol?
At the calling function side
At the called function side
TCP/IP
HTTP
POST
/ verb
HTTP/1.1
Host
Content-Type
Content-Length
Node.js streams
Node.js buffers
Request and response objects
Request
Response
Request and response life cycle
Request life cycle
'aborted'
'close'
'data'
'end'
Response life cycle

‘close’
‘finish’
Server configuration
‘maxHeadersCount’
‘timeout’
Server’s life cycle events
‘listening’
‘connect’
‘close’
Other networking APIs
HTTPS
HTTP 2
UDP
Conclusion

6. Major Web Server Components
Structure
Objective
Introduction
A static file server
Use case
Definition
Implementation
HTTP verbs (request methods)
Use case
Definition
Implementation
Route
Use case
Definition
Implementation
Endpoints
Cookie
Use case
Definition
Implementation
Cookie attributes

[Session](#)
- *Use case*
- *Definition*
- *Implementation*

[Request forwarding](#)
- *Use case*
- *Definition*
- *Implementation*

[Multipart form-data](#)
- *Use case*
- *Definition*
- *Implementation*

[Body parser](#)
- *Use case*
- *Definition*
- *Implementation*

[Cross-Origin Resource Sharing (CORS)](#)
- *Use case*
- *Definition*
- *Implementation*

[Dynamic web page](#)
- *Use case*
- *Definition*
- *Implementation*

[HTTP status codes](#)
- *Use case*
- *Definition*
- *Implementation*

[Server security](#)
- *Use case*
- *Definition*
- *Implementation*
  - *Data privacy and integrity*
  - *Cross-Site Scripting (XSS)*
  - *(Distributed or non-distributed) Denial of Service*
  - *Brute force (weak authentication)*

[Conclusion](#)

7. Interacting with Backend Components
- Structure
- Objective
- Backend components: internal services
  - *Intent*
  - *Design considerations*
    - *Functional*
    - *Performance*
    - *Reliability*
    - *Security*
    - *Serviceability*
- Backend components: external services
  - *Intent*
  - *Design considerations*
    - *Functional*
    - *Performance*
    - *Reliability*
    - *Security*
    - *Serviceability*
- Backend components: database
  - *Intent*
  - *Design considerations*
    - *Functional*
    - *Performance*
    - *Database end*
    - *Web server end*
    - *Reliability*
    - *Security*
  - *Serviceability*
- Conclusion

8. Implementing Common Website Features
- Structure
- Objective
- History of web pages
- Website – design considerations
  - *Usability*

- *Registered and unregistered views*
- *Navigation bar*
- *Articles*
- *Headers and footers*
- *Forms*
- *Search option*
- *Feedback forms*
- *Shopping carts*
- *Payment options*
- *Ease of use*
- *Consistent pages*
- *Short pages*
- *Short forms*
- *Home page*
- *Responsive web forms*

Website – elements and components
- *By language*
  - *HTML*
  - *CSS*
  - *JavaScript*
- *By elements*
  - *DOM*
  - *XMLHttpRequest*
  - *WebSocket*
  - *WebWorker*
- *By components*
  - *Hyperlink*
  - *Article*
  - *Dialog*
  - *Form*
  - *Options/dropdown*
  - *Table*
  - *Image*
  - *Audio*
  - *Video*
  - *Script*

Website: Advanced features

Google APIs
Embedding: Maps and social media
Pagination
Search
Filters
Geolocation indexing
Authentication
Admin dashboard
User profile
Conclusion

9. Making Our Website Production Grade
Structure
Objective
Enterprise enabling components
Performance
Reliability
Issue
Remediation
Issue
Remediation
Issue
Remediation
Issue
Remediation
Issue
Remediation
Issue
Remediation
Issue
Remediation
Issue
Remediation
Issue
Remediation
Issue
Remediation

- *Issue*
- *Remediation*
Availability
- *Issue*
- *Remediation*
- *Issue*
- *Remediation*
- *Issue*
- *Remediation*
- *Issue*
- *Remediation*
- *Issue*
- *Remediation*
- *Issue*
- *Remediation*
Scalability
- *Vertical scalability*
- *Issue*
- *Remediation*
- *Issue*
- *Remediation*
- *Horizontal scalability*
Observability
- *Trace (also known as log)*
- *Profiling/monitoring*
- *Dumping/snapshotting*
Security
- *Input validation*
- *Secure headers*
- *Secure sessions*
- *Authentication and authorization*
- *Data encryption*
- *Audit logging (aka logging for security)*
- *File system protection*
- *Shutdown unwanted ports*
Documentation
Conclusion

10. Best Practices for High Performant Code
Structure
Objective
Performance best practice: Hardware
CPU
Cache
Disk
Performance best practice: Network
Performance best practice: Operating system
Performance best practice: Runtime (Node.js)
Garbage collection
Event loop
Concurrency: "sync" versus "async"
Increasing concurrency
Remove debug options
Performance best practice: Application
Content optimization
Application decoupling
HTTP caching
API selection
Miscellaneous
Conclusion

11. Debugging Program Anomalies
Structure
Objective
Debugging crash
Preparation
Symptom
Useful data
Problem determination
Debugging low CPU
Preparation
Symptom
Useful data
Problem determination
Debugging high CPU

- *Preparation*
- *Symptom*
- *Useful data*
- *Problem determination*
- Debugging memory
  - *Preparation*
  - *Symptom*
  - *Useful data*
  - *Problem determination*
- Debugging performance degradation
  - *Preparation*
  - *Symptom*
  - *Useful data*
  - *Problem determination*
- Conclusion
- End

Index

CHAPTER 1
Getting Started with the Fundamentals

In this chapter, we will examine the basic premises of highly concurrent web workloads and introduce Node.js as an optimal platform for hosting such workloads. We will take a glance at the important characteristics of a web application and illustrate how Node.js defines a pioneering programming model that naturally aligns and resonates with these characteristics at the semantics level.

To do this, we will illustrate the basics of concepts like event-driven architecture, asynchronous programming, concurrency, parallelism, and scalability. Then, we will combine all these concepts to introduce the Node.js programming platform and showcase how it efficiently caters to highly concurrent web workloads.

Structure

In this chapter, we will cover the following topics:

- Introduction to event-driven architecture
- Introduction to asynchronous programming
- Concurrency versus parallelism
- Concurrency and scalability
- Putting everything together - Introduction to Node.js
- Core Node.js features

Objective

After studying this chapter, you should be able to understand the Node.js philosophy around event-driven architecture with asynchronous programming. You will also understand workload efficiency-related

concepts, performance characteristics of web workloads, and various tradeoffs that exist in resource usage versus performance. You will also learn some of the core Node.js features that are relevant to our goal of developing a web application.

Introduction to event-driven architecture

To better understand event-driven architecture, let's look at the existing programming models and traditional workload characteristics.

Calculator example

As we know, calculators were the primitive types of computers. How does a calculator work? We feed in the operands (values that are computed) and the operation. The device starts the calculation upon hitting a specific key ('=' or 'enter'). The activities performed by the calculator are as follows:

1. Receive the inputs - operands and operator
2. Parse the inputs – separate operands and the operator
3. Load the data and perform the operation
4. Store the data and / or display the result

The '*program code*' in this case corresponds to the four above-mentioned actions. Can you identify anything special with the code here? Once the program starts, it runs a set of instructions without any interaction with the '*programmer'* or user. In other words, the set of actions is run as a '*procedure'*.

Number sorting example

Let's take a more complex example. We have a list of numbers that needs to be sorted in ascending order. The most common steps carried out are:

1. Take the first number
2. Compare it with every other number in the list
3. If the given number is bigger swap the positions
4. Repeat these steps for every number

An actual working code written in JavaScript is as follows:

```
1. function sort(d) {
2.   for(var i = 0; i < d.length; i++) {
3.     for(var j = i + 1; j < d.length; j++) {
4.       if(d[i] > d[j]) {
5.         const temp = d[i]
6.         d[i] = d[j]
7.         d[j] = temp
8.       }
9.     }
10.   }
11. }
```

Do you see the same pattern here? Once the program starts with the loaded data, it runs a set of instructions without any interaction with the programmer or any other external program. So the set of actions is run as a procedure here as well.

Note: Procedural programming is a programming paradigm in which programs are connected series of computations steps. Calculations, function calls, loops, conditional branches etc. are standard building blocks that constitute procedural languages. First-generation languages such as Fortran, Cobol, and Pascal are classical examples of procedural languages.

Echo example

Now let’s take another example: a program that receives input from the user and echoes it back to the terminal. The steps carried out in this case are:

- Wait for user input
- Receive the user input in a variable
- Print the value of the variable
- Repeat the steps

A working example code (run with Node.js) is as follows:

```
1. process.stdin.on('data', (d) => {
2.   console.log(d.toString())
3. })
```

What happens to the program if the user is not supplying any data? The program has to wait till it receives something. Once it receives data, it completes an iteration and goes back to the waiting mode. The waiting period can be none, or it can be a few milliseconds to a few minutes to weeks or even infinite.

A notable difference between this program and the procedural programs that we saw earlier is that this program is driven by an event.

Web page access example

Another example is a program that makes a request with a remote service, waits for the response, and performs an action with the response. In this case, the response is obtained from a known entity (a website), but at an arbitrary time instance in the future, after the request has been made.

A working example code is as follows:

```
1. const h = require('http')
2. h.get('http://www.google.com', (r) => {
3.   r.on('data', (m) => {
4.     console.log(m.toString())
5.   })
6. })
```

Web server example

Yet another example is a web server that receives a request from a client and provides a response. In this case, the request is obtained from an unknown entity (a web client), at an arbitrary time instance in the future. The server is idle until it receives a request from some client.

A working example code is as follows:

```
1. const h = require('http')
2. h.createServer((q, r) => {
3.   r.end('hello world!')
4. }).listen(12000)
```

In the last three examples, the flow of the execution of the code depends on one or more external events. This is in due contrast with procedural programming, wherein the flow is fully dependent on its internal structure.

Event-driven programming is a programming paradigm in which the program flow critically depends on software-defined events. Programs that use contextual data in abundance are best suited for event-driven programming. A context can be a piece of data, occurrence of an event, time of event occurrence, or a combination of these. In the first example, the context is the user data, which is input. In the second example, the response from the remote service is the context and in the third example, the client request contains the context. In all three, the program responds to the events and works with the supplied context.

> **Note:** Event-driven programming is a paradigm in which the flow of the program is controlled by events, as opposed to self-directed code blocks. In such programs, events and contextual data pertinent to the event is vital to the logic of the program. Graphical applications, web servers and clients, and message-driven applications such as chat applications are best suited for event-driven programming.

Windows event loop

An example of an event-driven system is the classical Windows message loop that listens for mouse or keyboard events in **win32** applications. The message loop sits idle until an event occurs, and when it receives an event, runs a handler function to handle it.

```
1. while (true) {
2.   getMessage()
3.   handleMessage()
4. }
```

In the preceding pseudo-code, note that the '`getMessage()`' call can be potentially blocked until there is an event occurrence in the system.

Messaging example

Another example of an event-driven system is Kafka, which implements a message channel. Message publishers and subscribers communicate through the message channel, with message event as the primary trigger for actions.

```
1. // producer logic
2. producer.send(data, topic, callback)
3.
4. // consumer logic
5. consumer.subscribe(topic, callback)
```

Event-driven architecture is a paradigm in which connected components orient themselves around the lifecycle of software events.

Event-driven architecture is a perfect fit for web workload. We will examine the reason for this when we study other aspects of web workload and combine them together toward the end of this chapter.

Introduction to asynchronous programming

Asynchronous programming is a programming paradigm that defines program lifecycles outside of the main program flow. In other words, asynchronous programs implement out-of-order execution to help meet the main program's goal in an intermingled way. Easy visualization of an asynchronous program is two functions `foo` and `bar`, where:

- `bar` is executed while `foo` is in between its execution, and
- `bar` is not invoked directly from `foo`.

The seemingly complicated concept can be illustrated with a few examples.

Program interrupt example

Assume that you are running a program with a large loop. The program is taking more time than expected when run for testing purposes. You want to keep experimenting with the code, but how would you stop the program in the first place? The normal way is to wait for the program to complete, which will yield the terminal. In most terminal interfaces, we apply an interrupt (a key combination of Control and C) that stops the program in the middle of execution and terminates the program. How does this work?

The following actions occur under the cover in most computing systems:

1. Pressing of the key combination causes an interruption for the CPU.
2. The processor halts the current execution.
3. The processor stores the current execution context.
4. The processor consults with an interrupt vector table.
5. The table maintains a mapping between interrupts and their handlers.
6. The right entry is searched and located, and the handler is invoked.
7. The handler runs while the main program is halted.
8. The default action for a handler of *Ctrl* +*C* interrupt is to terminate the program.
9. So, the main program flow is abandoned and the program is terminated.

If the CPU logic around signal handling was implemented in software, an equivalent JavaScript code will look like this:

```
1. function handle(signal) {
2.   interruptVector.forEach((e) => {
3.     if (e.signal === signal)
4.       e.handler(signal)
5.   })
6. }
```

In this case, the handling of the keyboard interrupt is performed '*asynchronous*' to the long running loop. In addition, this '*asynchrony*' is required for the proper functioning of our program.

In this example, the main flow is terminated due to the asynchronous execution.

Multimedia example

Let's examine another example. You are playing a movie from your favorite site. A multi-media player software runs in your browser and downloads data from the vendor's site, stores it in an internal buffer, synchronizes the audio and video, applies the play configuration such as speed, quality, and so on, and then renders the movie in the player's client area. While the movie is on, after some time, you want to pause the play. When you click on the pause button, these things happen:

1. Playing of the video content is stopped
2. Playing of the audio content is stopped
3. Data downloading is continued until an internal buffer is full

The sequence of actions in a **`pause()`** function can be represented as follows:

```
1. function pause() {
2.   video.stream.pause()
3.   audio.stream.pause()
4.   if (buf.length === MAX)
5.     downloader.pause()
6. }
```

These things are performed '*asynchronous'* to the playing sequence.

In this example, the main flow is not terminated due to the asynchronous execution but only suspended for a duration and then resumed from the point where it was suspended.

JavaScript web page example

The most common example is of the '**`onClick`**' and '**`onSubmit`**' handlers in client-side JavaScript, which performs actions asynchronously.

```
1. <html>
```

```
2. <body>
3. <script>
4. function foo() {
5.   alert("hello world!");
6. }
7. </script>
8. <form action="p8.html" onsubmit="foo()">
9. <input type="text">
10. <input type="submit">
11. </form>
12. </body>
13. </html>
```

From all these examples, it is evident that there are common programming scenarios where one or more actions need to run out-of-order with the main program flow. It is not meaningful for the main program flow to cater to these out-of-order actions without severely distorting the code or losing precision. So, an established design pattern is to let the main flow focus on the core logic while the asynchronous handlers focus on the out-of-order actions.

Operating system scheduler example

Operating system programs are filled with asynchronous actions. Process scheduler comes into action at a predefined time interval, de-schedules a currently running process, and schedules a currently waiting process. It is a classical use case where asynchronous programming is at its best.

```
1. setInterval(() => {
2.   cpus.forEach((c) => {
3.     c.running.process.runTime += 10
4.     c.running.process.state = 'runnable'
5.     c.runnable.push(c.running.process)
6.     const newProcess = c.runnable.shift()
```

```
7.      c.running.push(newProcess)
8.      c.running.process = newProcess
9.      c.running.process.state = 'running'
10.   })
11. }, 10)
```

One of the important aspects of asynchronous programming is the way they orchestrate the main and asynchronous flow of execution. In other words, how the main program flow is suspended temporarily, how its contextual data is preserved, how a new execution environment is created where the asynchronous code runs, how both the flows communicate if need be, and finally, how the main flow comes back to life from the contextual data, if need be.

Asynchronous programming is a natural fit for web workload. We will examine the reason when we study other aspects of the web workload, and combine them together, towards the end of this chapter.

> **Tip**: Asynchronous programming does not necessarily involve multiple threads. A piece of code B is executed asynchronously with another piece of code A if B is executed irrespective of the completion status of A, and vice versa. That is, one code does not need to wait for the other to complete.

Introduction to functional programming

Functional programming is a programming paradigm in which function definitions are trees of expressions – the vortex of each tree is a function. The most relevant feature of functional programming in our context is that functions are treated as first class variables.

This means functions can be passed as arguments to other function calls, can be returned from calls, and can be assigned to and assigned with values.

Function passed as a value

An example in which function is passed as a variable:

```
1. function foo() {
```

```
2.   return 10
3. }
4.
5. function bar(fn) {
6.   return fn() + 10
7. }
8.
9. bar(foo)
```

Function returned as a result

An example in which function is returned as a result:

```
1. function foo() {
2.   return 10
3. }
4.
5. function bar() {
6.   return foo
7. }
8.
9. const ret = bar()
10. ret()
```

Function as assignment subject

An example in which function is assigned to variable, and function is assigned with a value:

```
1. function foo() {
2.   return 10
3. }
4.
5. const bar = foo
```

```
6. foo = 10
```

An implied aspect of treating functions as first-class variables is the ability to define functions inside a function, just like we define data elements inside a function.

Closure functions

```
1. function outer(op) {
2.   let ol = 1
3.   function inner(ip) {
4.   let il = 2
5.     // access to ip and il as always
6.     // access to op an ol even after outer exits
7.   }
8.   return inner
9. }
```

In the previous example, ‘outer’ is a function that defines (or embed) another function ‘inner.’ By definition, such a method is called a Closure function. Among other things, the most important aspect of a Closure is that, at the time of its definition (when ‘outer’ is invoked and the control flow reaches line 3 in the above-mentioned code) a Closure context is created with the ‘inner’ function defined in it, alongside the values of the variables that it can access. These are:

- **ip:** Its own parameters
- **il:** Its local variables
- **op:** The embedding function's parameters
- **ol:** The embedding function's local variables

But does invocation of 'outer' lead to invocation of 'inner'? No. The Closure function is assigned to a variable as return value to the call to 'outer'. After that, 'inner' can be invoked like any other function.

The following example illustrates this fact:

```
function outer() {
  let ol = 1
  function inner() {
    // o1 is retained for inner
    console.log(ol)
  }
  return inner
}
const foo = outer()
foo()
```

In this example, the local variable of ‘outer’ is retained in the closure context of ‘inner’ for its own use, whenever ‘inner’ is invoked.

Closure context

In this model, when the function 'inner' is invoked, the function 'outer' would have been returned long back, and will not be in the scope. So the question is, how are the variables 'op' and 'ol' available to 'inner'? When 'inner' is created, it just does not create a normal function. Instead, the function and its defining environment (the closure context) is also created and bound to that Closure function. In this case, the current values of 'op' and 'ol' at the time of defining 'inner' also get pinned to the closure, detached from the 'outer' function.

What is the big deal about these seemingly ‘ugly’ constructs? What great capabilities do they add to general purpose programming? Do they improve or damage readability and purity of the code? Again, the relevance of functional programming, more specifically Closure, will become apparent when we combine all the concepts together at the end of this chapter.

> **Tip**: In the above example, if the ‘outer’ function is called multiple times with potentially different arguments, as many Closures will be created and returned, with unique Closure contexts associated with each.

Concurrency

Concurrency in programming is execution of multiple tasks coherently but interweavingly. In our definition of concurrency, the tasks are logically related but physically not necessarily. Again in our definition of concurrency, the tasks are executed interweavingly, but this is not a necessity in general definition. We will examine these aspects with the help of multiple examples:

Cooking example

Think of a chef working in a kitchen, making pasta. The following is a rough sequence; the exact steps may vary based on the recipe, but that is not important here:

1. First boil the pasta noodles / sticks for 5-10 minutes.
2. While that is happening, get vegetables such as onion, garlic, bell pepper, and so on.
3. Cut them into fine pieces.
4. Rinse the pasta to be boiled.
5. Gather the necessary sausages.
6. Check if the pasta is boiled and drain the water.
7. Deep fry the vegetables in oil and add salt to the fried veggies.
8. Add the boiled pasta to the fried vegetables.
9. Add sausages.
10. Stir for a few minutes.
11. Leave it for 1-2 minutes.
12. Meanwhile, put the rest of the vegetables and sausages back to their places.

Let's see how various activities will look if plotted on a time graph:

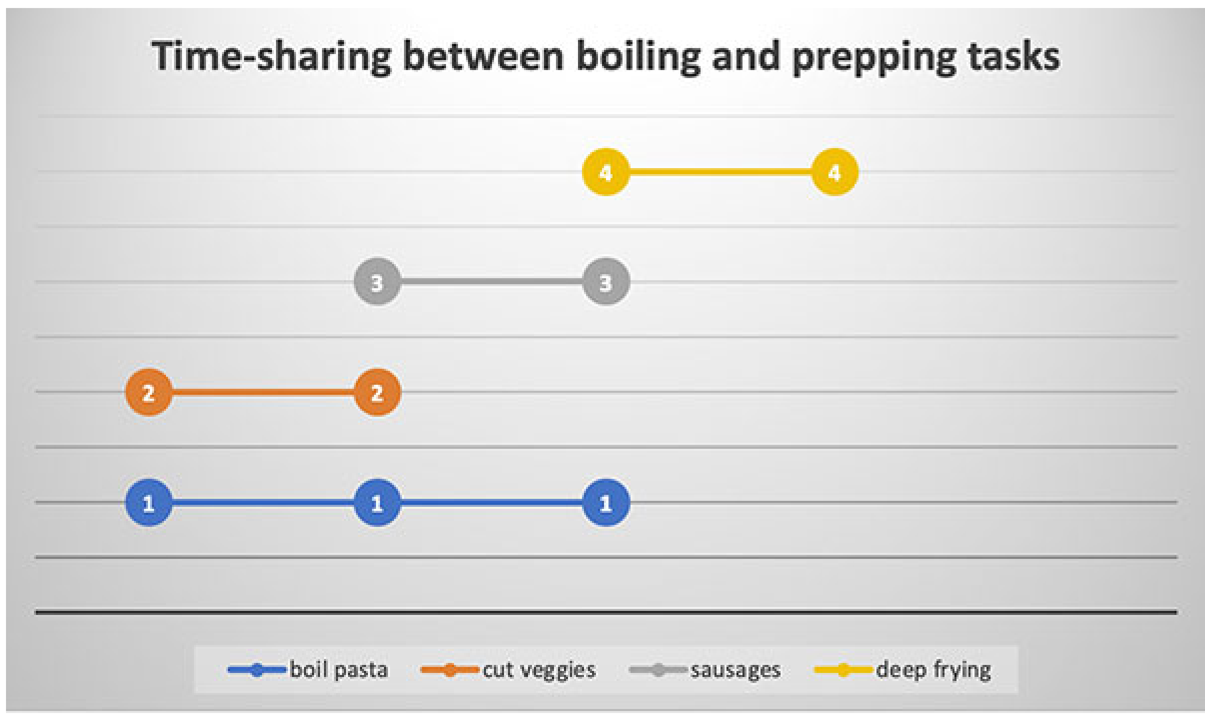

Figure 1.1: *Concurrency with time sharing - I*

In the above graph, the lines represent an action that runs for a period of time. The presence of two lines in the same time frame indicates actions that are taking place simultaneously.

Here, the key part of concurrency is when the chef was doing multiple things (items marked in bold) since they keep the pasta noodles to boil. How many items were executed in that time frame? How were they executed? How are those items related? Revisiting the definition of concurrency in this context, it is the execution of multiple (cooking) tasks coherently, wherein the tasks are logically related and executed in an interweaving manner.

Progress bar example

In our next example, let's see what happens when we download a large file from the Internet. A progress bar shows how much of the file is downloaded, how much is remaining, and where does it stand in a graphical manner; it is called the **progress bar**. Let's see how it works:

1. Calculate the total bytes to download: b.
2. Identify the number of blocks in the progress bar: n.

3. Compute the number of bytes that correspond to one block: b/n.
4. Start downloading the data.
5. Compute the current total every time data arrives: i.
6. Compute current progress p = i % n.
7. Render / ensure p blocks are marked in the bar.
8. Resume the download and repeat the steps.

Again, let's see how the two activities will look if plotted on a time graph:

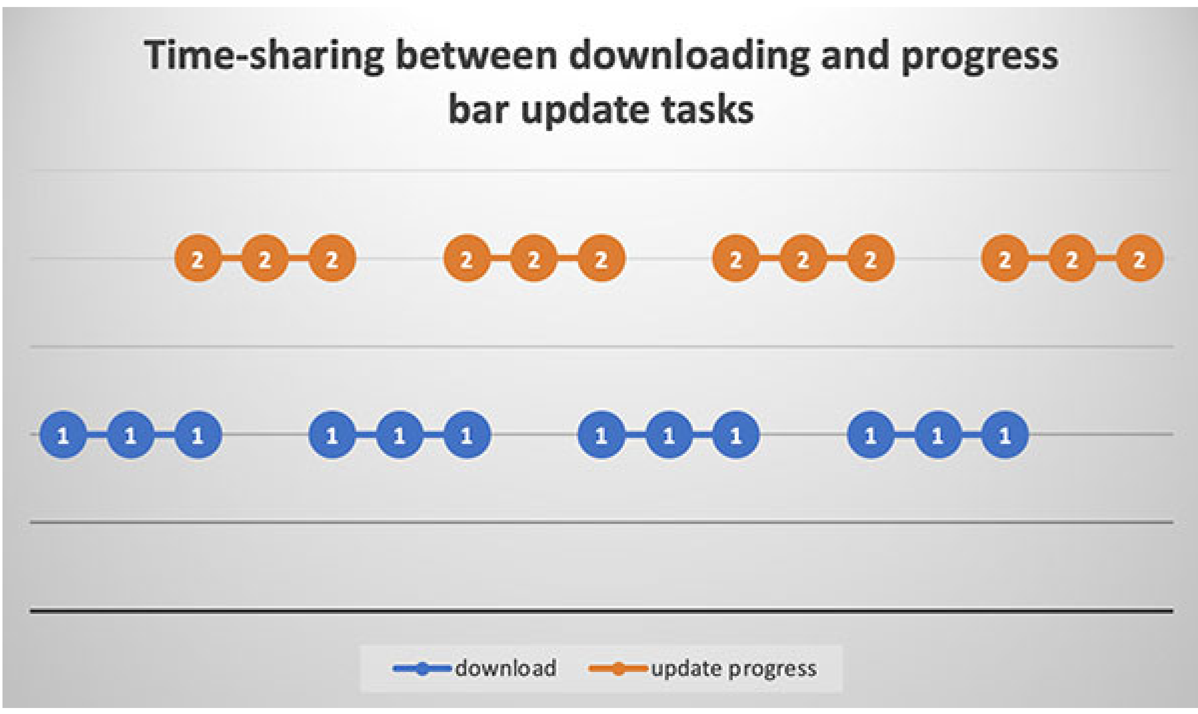

Figure 1.2: *Concurrency with time sharing - II*

Here, the key part of concurrency is between the data download, recalculating the progress, and refreshing the progress bar. How many items were executed in that time frame? How were they executed? How are those items related? Revisiting the definition of concurrency in this context, it is the execution of multiple (downloading) tasks coherently, wherein the tasks are logically related and executed interweavingly.

Garbage collection

In a more complex example, many modern language runtimes employ a technique called garbage collection, wherein the runtime heap is

enumerated and stale objects (program data) are purged occasionally. This activity is performed by utilizing a small amount of CPU time from the application execution without hampering its performance or showing any visible pause. We say that the garbage collection runs concurrent to the application while garbage collection and application execution are logically related but physically not, and the actions are executed interweavingly.

Note: Concurrency in a nutshell is multi-tasking with time sharing by a single resource.

Parallelism

Parallelism in programming is the execution of multiple tasks truly independently and parallelly. The tasks may be logically related or unrelated but are executed with physical separation between them. The tasks are executed independently. We will examine these aspects with the help of multiple examples:

Consider the previous example of making pasta. If this were part of a large restaurant with lot of tables and rush in the busy hours, there would be many chefs making pasta independent of each other. Furthermore, there would be chefs who make other food items as well.

Again, let’s take a look at the time graph:

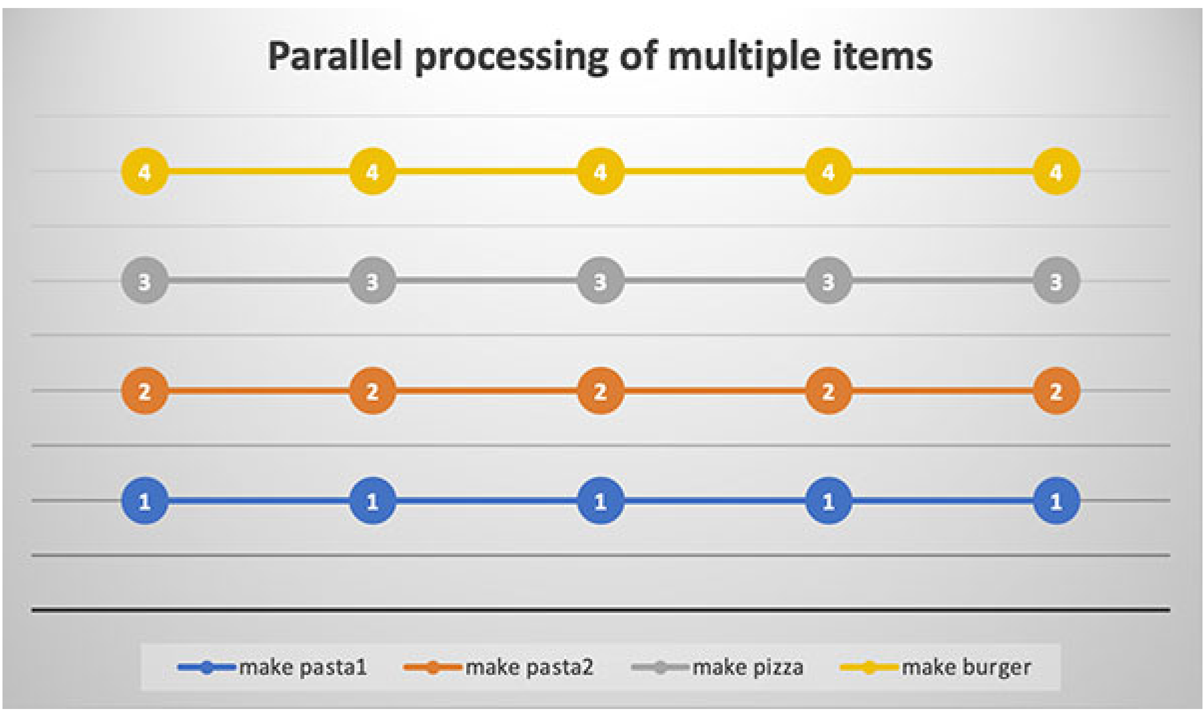

Figure 1.3: *Concurrency without time sharing*

Distributed word counting example

In a more complex example in programming, let's take the classical word count problem in a massive list of a billion records. For this, the algorithm may run as follows:

- Identify the number of parallel tasks we want to run: n.
- Split the record into n equal records.
- Assign each record to each parallel task.
- In each task, iterate over the list, search for the word, and make a count.
- Iterate over each task and add up the word count that each task gathered.

In summary, we implement parallelism by executing similar or dissimilar tasks independent of each other. Parallelism improves application performance.

> **Note:** Parallelism, in a nutshell, is multi-tasking without time sharing by multiple resources.

Concurrency versus parallelism

Now that we understood concurrency and parallelism separately, we will compare the two concepts and discuss the similarities and differences between them. This distinction will be of utmost importance when we understand certain principles of Node.js.

Similarity

- In both, multiple tasks progress together
- In both, program responsiveness improves

Differences

- In concurrent, only one task **runs** at a time, while in parallel, multiple tasks run together.
- In concurrent, one resource switches between tasks, while in parallel, each task has a resource.

In a real program, a task would essentially mean a block of code, and a resource would mean a processor that runs a block of code.

If you revisit the two concepts with this in mind, we can infer that:

- In concurrent, a CPU time-slices between multiple code blocks
- In parallel, multiple CPUs run multiple code blocks

It is immaterial whether or not the multiple code blocks are part of the same program, at least in the context of our discussion.

In higher-level programs, a CPU is abstracted as a **Thread**. Refining our differentiation further, we can say that multiple threads run parallelly while a single thread switching between tasks runs concurrently.

- Why do we want to switch tasks when you have more threads in the system?
- Which one will be more efficient - a thread multiplexing between ‘n’ tasks or ‘n’ threads running n tasks in parallel?

- What is the overhead of switching between tasks in a single thread?
- When is it appropriate to switch a task?
- How can you resume/return to the first task after switching from one task to another?

Think about these questions and try to answer them. We will answer these in the next section.

Concurrency and scalability

Let's discuss our own case here. When a web server is serving many clients, how does it manage all the connections together? If the server handles client requests sequentially, the subsequent clients would wait and eventually, timeout depends on the number of simultaneous requests. This is a case of poor scalability of the server.

Thread pooling

Traditionally, server implementations used to solve this problem by implementing two types of scalability: vertical and horizontal. In vertical scalability, a number of threads (roughly equal to the number of cores in the system) are created, prepared, and managed in a farm or a pool, called a **thread pool**. Each client request, as and when it arrives, will be picked up by one free thread from the pool, and that thread serves the client to its completion. This solves the problem of poor scalability.

Horizontal scaling

What happens if the number of simultaneous client requests is much higher than the number of cores in the server hardware? Then the pool starves, and the scenario goes back to the first case – client requests get queued up and eventually time out. In such scenarios, more machines are provisioned to run the same server application, and the requests are distributed to these servers. It brings back the desired scalability. This is called horizontal scaling.

What is the tradeoff here? In vertical scalability, we exploit the multicore by parallelizing request handling. However, each thread comes with its

associated resource consumption – such as thread stack, working memory, context switch, scheduling, lock contention, and so on. Similarly, in horizontal scalability, we replicate the server in multiple systems to further parallelize request handling. Naturally, each hardware implies more resource consumption.

In summary, we trade additional resources to improve scalability and responsiveness.

Can we condense more requests in one server system itself without losing responsiveness and adding more resource pressure? What if we increase concurrency to our server rather than parallelism? As per definition, a concurrent program does multi-tasking with a single thread. So, can a thread switch between multiple client requests and serve them in an interweaving manner?

Answering this question will take us directly to the principles of Node.js.

Introduction to Node.js

Now that we learned a number of seemingly unrelated concepts, let's stich those together to understand the underpinning theory behind Node.js.

Latency in computer systems

Have you thought how much time different operations take in your computer? It depends on the operation itself. The following chart gives a rough estimate on how much time it takes for a unit amount of data to be moved from one endpoint to another. An endpoint here is any recordable media in a computing system.

Endpoints	Latency (in nano seconds)
Registers	~0 (reciprocal of clock frequency)
Hardware data cache	10
Main memory	100
Secondary memory	10000000
Network	15000000

Table 1.1: *Latency of data transfer between endpoints*

What does this mean? By virtue of the way some of these devices (endpoints) are manufactured, installed, and operated and due to some cost versus performance tradeoffs that the vendors have opted while designing these devices, data movement in some endpoints can be faster, while some can be slower. For example, registers and main memory are part of the processor board itself. External disk requires interaction with an external software as well as physical movement of the disk head. In other words, the registers and memory are bound to the CPU, whereas the disk and network require input and output operations with the controlling hardware and software. Operations involving input and output devices are generally called I/O operation, and hence, such operations are said to be bound to I/O.

This means an arithmetic operation (such as adding two numbers) and storing the result will be much faster than fetching a similar data from the disk. And needless to say, the difference in latencies of those two types of operation is huge and is not comparable.

CPU bound and I/O bound operations

Operations involving the CPU bound devices - first three endpoints in the above table, i.e., registers, cache, and main memory - are called CPU bound operations. Operations involving, I/O bound devices - secondary memory (disk) and network - are called I/O bound operations.

Tip: A CPU bound operation can typically be a million times faster than an I/O bound operation.

There is a special case to be mentioned here. The latency comparison is based on an assumption that the endpoints are ready to send and receive the data in question. This is always the case in CPU bound operations, while it is not so in I/O bound operations. What if the rate at which the server is writing is higher than the rate at which the client is reading, and the network transport buffer in the operating system kernel becomes full? What if a client is not reading at the time the server is writing? Depending on how the endpoints are designed to function, an initiated data transport can occur now, later, or never! The latency values shown in the previous table are measured when both the endpoints are ready to transfer data.

If an endpoint has initiated a data transfer (read / write) and the other endpoint is either not ready or incurs visible latency, the owning thread that initiated the data transfer will be seen as blocked on I/O. A thread waiting for I/O completion will be mostly be de-scheduled by the operating system scheduler, and the owning CPU will be used to run some other program, rendering the said thread in the current process as dormant.

Characteristics of web workload

A scientific computation is predominantly a CPU bound application, as it involves lot of operations on data that is already residing in the registers or main memory. A chat application, on the other hand, is an I/O bound one, as it involves a lot of operations on data that is either incoming from the network or going out to it. A web server is a fine mixture of CPU bound and I/O bound work. Every connected client requires at least two I/O operations (one inbound request and one outbound response). If the client makes multiple requests, the number of I/Os multiplies too. In addition, the server would perform some operations between the request and the response, which can potentially be CPU bound activities or even I/O operations.

In summary, a fully loaded web server will contain numerous I/O bound activities.

Bringing the context of scalability

Dissecting further, if you profile a single request in a web server, you can see that it:

- Accepts a client connection
- Reads the client request
- Parses the client request
- Performs some computation
- Prepares a response
- Writes back to the client
- Closes the connection

Among these seven operations the server performs, the first two and the last two items are I/O bound. Assume that it takes 1 millisecond for the whole sequence to complete. How will the division of time among these tasks look like? Given the latency theory at the beginning of this section, it would be like 99.9999:0.0001 or even more skewed.

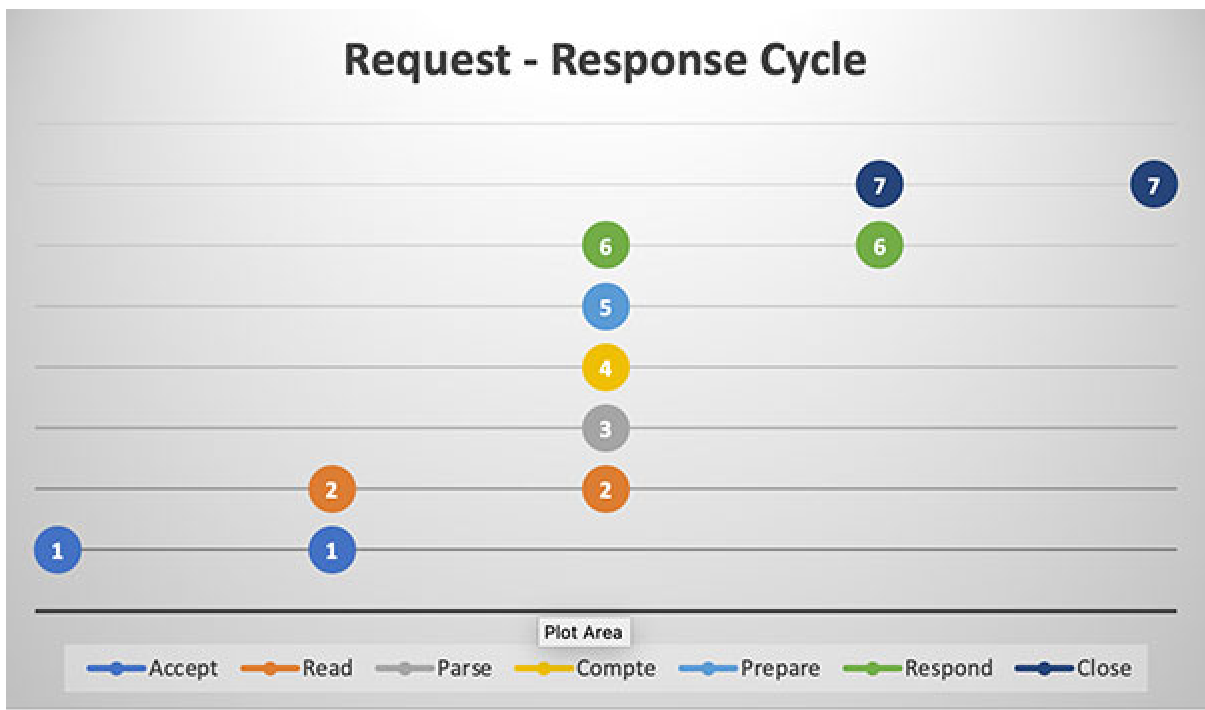

Figure 1.4: *Division of tasks between a Request-Response cycle*

This means if you take a peek at the thread that handles the request at a representative time instance, it might be waiting – either for the input or for writing the output. So, due to the presence of rich I/O, even when the server is fully engaged, we observe less than 20% CPU utilization in a typical web workload.

So now we have identified two issues:

- A single thread is not sufficient to handle simultaneous client requests (scalability issue that we discussed earlier)
- A single thread itself is under-utilized for most of its service time – the request response phase (due to the high presence of I/O operations that takes longer as compared to computations)

The first issue can be solved with thread pooling—again as we discussed earlier, with its own tradeoff—of proportionately high resource

consumption. The second one is a new finding. Do things improve if we correlate the two issues and synthesize new ways of running our workload?

On the one hand, a single thread is not enough for handling all the load, and on the other hand, a single thread itself is idle for most of the time! The thread's idle time could be leveraged to run other processes in the system – but in a production-grade server computer, there aren't many other critical processes than the server process itself. So that is not a great relief and does not provide much benefit.

What if the idle thread is made to handle another waiting client? (Harp on this question. If this has occurred to you already, give yourself a pat on the back! This thought is exactly the pivot of innovation around Node.js technology; the rest is implementation detail.)

Bringing back the context of asynchronous programming

Yes, that is possible. In the previous sections, we have seen a case wherein a running program is interrupted asynchronously, performing an unrelated task and coming back to resume the main program flow. What if we reuse that technique here? That is:

- Process n^{th} client request until it reaches a slow I/O operation
- Detach the thread from nth request
- Attach the thread to n+1th request
- Process $n+1^{th}$ request until it is either complete or reaches a slow I/O by itself
- Revisit n^{th} request to see if that can move ahead now
- If yes, repeat the logic in a cycle
- If not, go for a third request, $n+2^{nd}$ request, and repeat the cycle

In short, we can run the single thread asynchronously between multiple client requests, much like we ran an asynchronous interrupt signal or controlled a multimedia stream in previous examples, with these main differences:

- Asynchrony is triggered internally as opposed to externally

- Asynchrony is triggered in response to encountering slow I/O

The asynchrony comes in by way of running different blocks of code concurrently, much like we cooked pasta. downloaded a huge file with a progress bar, or ran garbage collection concurrently with the main program flow. The key points to note are:

- A single thread multiplexes between many client requests
- A single thread advances to the extent it can, regardless of the previous work
- No differentiation between main and side car program flow – all are main flows

What would we achieve with these adjustments in place? We can execute a million CPU bound operations instead of a single blocking I/O if the multiplexing is absolute – this is because that is the I/O latency - CPU latency ratio. And it is a huge benefit! But does that mean we can handle a million clients concurrently? Probably not. Each task will require some CPU bound processing as part of the request handing. So the benefit of concurrency will be lesser than that.

However, compared to the existing alternatives, this mechanism provides a great level of concurrency. As the thread is no idler, the switching happens in-process and in-thread, and each task is able to run to its potential. At any given point in time, no task waits to get resources allocated, and none has any additional resource allocation!

So far so good. Now, we have to address two problems to achieve this concurrent task execution:

- Manage the I/O multiplexing
- Manage the context switching

Each of these items is discussed in separate sections.

Bring back the context of event-driven architecture

Remember the definition of event-driven architecture? A paradigm in which the program flow is controlled by events. In the context of the concurrent

execution, we need events. More specifically, we need events that will trigger a change in program flows. And the main program flow changes from our algorithm are:

- Encountering an I/O, needing current flow to pause and a new flow to come in
- Encountering an I/O completion, needing the old flow to resume

The core of Node.js architecture is to address exactly these two aspects. Using the native terms of Node.js, this would translate to:

- Convert blocking I/O as non-blocking I/O
- Create and push an event pertinent to I/O
- Continue with the rest of the code
- When I/O completes, trigger the matching event
- Handle the event by invoking its continuation handler

I/O multiplexing becomes possible by customizing the single thread in this way. Many operating systems provide means to convert blocking I/O to non-blocking ones and allow one to later 'poll' the system to see if the I/O has been completed. Similarly, as we are storing the context of I/O initiation, it is possible to invoke the completion handler/continuation handler after the I/O completion.

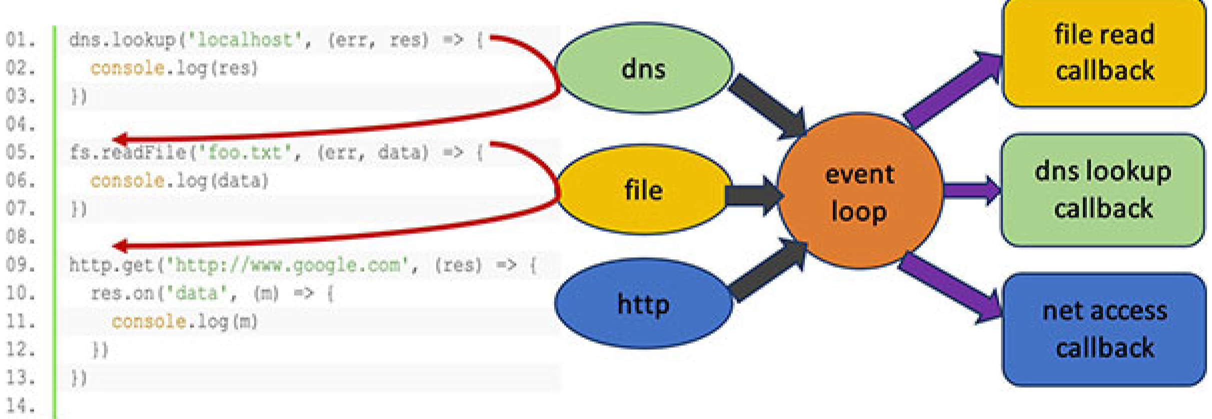

Figure 1.5: *Node.js' Event-driven architecture*

In this architecture diagram, there are blocks of code in the left side pane: A 'dns' lookup, a file read operation, and a network access – all of which take a lot of I/O cycles and can potentially block the thread. In Node.js, the API

abstractions that deal with these operations are designed in such a manner that they don’t block. Instead, they initiate a non-blocking I/O after creating an event and binding it with the completion handler function. The thread returns immediately, as shown with the red arrows. When each event actually gets triggered, the event is picked up and the binding function is identified and invoked with the result of the I/O operation as the input. This is repeated until all the events are processed.

The main component that manages the event lifecycle is called **event loop**.

With this setup in place, the events are processed at exactly the time when they occur. In addition, the thread is not waiting for an I/O to complete. This solves both our above-mentioned problems: issues with many threads and issue with the thread blocking on I/O. When the I/O become non-blocking, the thread is continuously engaged and is executing many tasks in a time-shared manner, wherein the slicing is decided by occurrence of the events. Now if we profile the Node.js single thread in a reasonably loaded server application, we see that its average CPU consumption is above 90% - an indication of highly efficient concurrency. This is a perfect fit for the web workloads.

At this point, we are ready to revisit all the questions we encountered earlier and attempt to explain those in the new architecture’s context.

Why do we want to switch tasks when you have more threads in the system?

This is because:

- Threads are expensive – they consume resources
- Threads are underutilized on I/O workloads

So, if the workload contains homogenous transactional types (request-response cycle in the web server example), and those transactions contain heavy I/O, then switching tasks using a single thread is beneficial both in terms of resource consumption and CPU utilization.

Which method will be more efficient: a thread multiplexing between ‘n’ tasks or ‘n’ threads running n tasks in parallel?

If efficiency here is a measure of the tasks performed per resource consumption, then, as illustrated with the architecture, a thread multiplexing between ‘n’ tasks is much more efficient than the other one.

The single thread is seen as:

- Utilizing most of its allocated CPU share
- Responding to I/O events as and when they occur
- Handling multiple client requests simultaneously

What is the overhead of switching between tasks in a single thread?

Bring back the context of functional programming

Generally, the overhead is to preserve the context of the outgoing task and resume the context of the incoming task. In Node.js, the context preservation and resumption are achieved by using the JavaScript primitive called **Closure**, which we learned earlier under the header functional programming.

```
1. function foo(p1) {
2.   let l١ = 10
3.
4.   function bar(p2) {
5.     // access p1, l1 and p2
6.     // from the original context
7.   }
8.
9.   anAsyncIOMethod(i١, bar)
10. }
```

Following the definition of Closure functions, in the example above, '**`foo`**' is a function that embeds a Closure function '**`bar`**'. Among other things, the most important aspect of a Closure is that at the time of its definition (when '**`foo`**' is invoked and the control flow reaches line 4 in the code above, a Closure context is created with function '**`bar`**' is defined in it alongside the values of the variables in the outer function '**`foo`**'. When '**`bar`**' is actually invoked, it has access to its own parameters (p2), the local variable of its outer function (l1) as well as its parameters (p1). So assuming an IO routine is invoked through '**`anAsyncIOMethod`**', which registers an IO event with

the event loop and returns immediately, it will resume its continuation through ‘`bar`’ when the IO actually completes. Now because ‘`bar`’ is a Closure function, it ‘*remembers*’ the context in which it was created. When ‘bar’ is thus invoked by the event machinery, it gets access to the ‘old’ context in which the asynchronous method was invoked.

In summary, Closure functions provide a powerful building block for implementing event-driven architecture in Node.js, alongside asynchronous programming. They are largely used as completion handler functions or continuation functions for asynchronous methods.

When is it appropriate to switch a task?

It is appropriate when the currently running task has exhausted all of its activities or succumbed to an asynchronous (I/O bound) activity. After initiating the I/O action and registering for its completion, the thread switches to the next task. In reality, the asynchronous call just returns as if it is completed, and the following code is executed. Later, when the asynchronous activity completes, the associated completion handler (typically a closure function) gets invoked.

Is it mandatory that the completion handler be a closure function?

No. We want the completion handler to be a closure function only if we want to resume the original context in which the asynchronous method was invoked earlier (as closure functions remember its environment). If the completion handler executes a pure code that does not have dependency on any other part of the system, it can be a regular function too.

Core Node.js features

Now that we understand the basic premises of asynchronous programming with event-driven architecture that Node.js pioneers, let’s quickly look at some of the notable and interesting features of this platform. We will expand many of these aspects later. This section serves as a quick-byte experience for its core features.

Structure

The core interpreter that provides a JavaScript runtime environment for Node.js is ‘v8’. This component is responsible for executing general

purpose JavaScript code. The event machinery is called event loop or '`libuv`' (as the name of the project) that is written in ANSI C. There are several APIs (notably asynchronous APIs) that are modules or functions that provide a pure JavaScript interface to the programmer. Under the cover, these APIs interact with the event loop through a C++ intermediary called "wraps". Node.js is written in C++ that integrates these components together and orchestrates the asynchronous event-driven JavaScript programming.

APIs

Many Node.js APIs are asynchronous by specification. For most of these asynchronous APIs, synchronous counterparts are also provided – for convenience to certain user programs that do not follow the philosophy of asynchrony.

Many APIs are event emitters. This means many objects that are part of these APIs are capable of managing events. This is an essential Node.js abstraction that helps to deal with the event machinery through JavaScript semantics.

Note: The famous ".on" primitive of event emitter class technically and philosophically symbolizes the event-driven architecture – it's highly expressive semantics tells: "Upon an event occurrence on this object, call that function". Many APIs that implement event handling, inherit from event emitter Class.

Few APIs abstract common base capabilities such as operating system calls, in a platform independent manner. As we know, abstracting native capabilities is important for usability of non-native languages.

Streams

Streams are just another API but require a special mention here. By definition, a stream is flowing data. In the context of event-driven architecture, the inception, segregation, aggregation, transportation, and exportation of program data is much more efficient if the data is in a flowing manner as opposed to static. This is because several endpoints that take part in the program data processing may be at different lifecycle phases

and implementing code blocks that are inherently capable of working with flowing data provides optimal performance. Many APIs that deal with disk and network I/O employ streams to cater to their main data transport and management. There are readable, writeable, duplex, and transformer streams defined as part of the API specification. Streams are event emitters and have a long list of lifecycle events associated with them.

Small core philosophy

One of the reasons (other than high performance for I/O workloads) why Node.js platform is highly successful is its small footprint and fast startup that makes it a natural selection for Containerized workloads. This is achieved by:

- Maintaining the core Node.js to be as small as possible
- Building all the dependent code into a single binary
- Implementing only the most common capabilities in core

This means there is no separate enterprise edition or standard edition for Node.js; all we have is a single, simple core set of APIs. Any extra requirements are satisfied by userland extensions called '`npm`' modules.

> **Note:** The Node.js APIs provide well defined and high-level abstraction on top of the low-level networking capabilities such as TCP, HTTP, File system etc. Hence, these are termed as core capabilities. The extended capabilities that build on top of the core ones are usually implemented as a library outside of the Node.js core and termed as "userland".

npm

NPM stands for **Node Package Manager**. It is a built-in component that defines the management capabilities for reusable node.js libraries and modules, also called as `npm` modules. The small core in conjunction with the ability to integrate with reusable modules significantly improves the developer productivity of Node.js applications. At the time of this writing, npm registry hosts the world's largest software library.

Conclusion

In this chapter, we learned the basic premises of highly concurrent web workloads and the asynchronous, event-driven programming of Node.js. We also understood how this programming style is a natural fit for the said type of workloads. To understand the programming style, we learned a number of concepts such as concurrency, parallelism, scalability, functional programming etc. and then combined these concepts to learn the typical performance characteristics of web workload in the context of these entities. Then we learned how separating CPU bound and I/O bound tasks and multiplexing between multiple tasks while making I/O bound tasks non-blocking and event-based yields a high level of performance. We understood this as the key philosophy behind Node.js architecture. We also looked at some of the peculiar features of Node.js that will be useful when we design our web server application.

In the next chapter, we will describe the essential pre-requisites for running the web application that we develop, enabling us to kick-start with our web server development work. While these pre-requisites are suitable for the development stage, many of these will go forward for considerations in the production stage as well.

Points to remember

- CPU bound operations are much faster than I/O bound ones
- Web workload - both client and server - are I/O intensive
- Traditional blocking I/O behavior cost a lot of resources to web programs
- Node.js converts blocking I/O to non-blocking I/O
- Node.js initiates and pushes I/O tasks asynchronously and switches tasks
- JavaScript and its functional semantics are conveniently chosen to compliment event-driven architecture
- With this model in place, Node.js provides unprecedented scaling density for web programs
- Node.js leverages 'v8' JavaScript engine to execute JavaScript code

- Node.js leverages '**`libuv`**' to manage event lifecycles
- Node.js follows '*small core*' philosophy with large number of third-party reusable modules that provide common extension capabilities
- '**`npm`**', Node.js' in-built package manager, facilitates seamless integration of Node.js applications with the said reusable modules

CHAPTER 2
Setting Up the Environment

In this chapter, we will describe the essential pre-requisites for running the web application that we develop. Here, we aim to build a web server from scratch by leveraging only the Node.js platform. There are not many pre-requisites, but we still need to make subtle decisions that we need to make in order to be running our application as a stable web server capable of running real workload. While these decisions are suitable for the development stage, many of these can work for considerations in production stage as well. This is mostly because Node.js does not have separate editions for standard and enterprise applications.

Structure

In this chapter, we will cover the following topics. In general, they cover selecting:

- The application development platform
- Node.js versions
- Dependencies
- Resource requirements
- Code editor

Objective

After studying this chapter, you should be able to understand the pre-requisites for developing a Node.js application from scratch. This will include understanding the supported platforms, common development tools and dependencies, selection criteria for Node.js version, its resource usage, and so on.

Platform selection

There are a number of platforms (14 at the time of writing this book) that support Node.js. The tier1 platforms are Linux, Windows, and MacOS, which means these are platforms where Node.js release line is maintained with the best focus. So, we will only pick up those in our discussion.

Linux and Windows operating systems are commonly used operating systems and hence, do not require any explanation. MacOS is a UNIX variant that brings in the best of command-line programming and operating system semantics of UNIX and graphical user experience of Microsoft Windows. While all three are equally good to meet the objective of this book, if there is a personal choice to be expressed, we recommend MacOS for the development of our application due to enhanced user experience with its interfaces.

There is very little difference between these platforms with respect to the Node.js API abstractions. For example: signals, path separator, file permissions, user resource limit settings work differently in Node.js on UNIX and Windows. However, these are emanating from the underlying platform's differences itself and hence are trivial and very familiar to the developers already. For example, Windows uses backward slash ('\') as path separator, while MacOS and Linux use forward slash ('/'). These fundamental differences are well explained in the Node.js API documentation at the respective pages of the API and illustrated with examples.

Tip: A full list of supported platforms (operating system and underlying architecture), kernel and C runtime versions, and support tier structure are listed here:

https://github.com/nodejs/node/blob/master/BUILDING.md#platform-list

Detailed descriptions and additional platform-specific instructions are also provided on the same page.

Node.js version selection

Node.js versions with even numbered major versions are **Long-term Supported** (**LTS**) variants. For example, v10.x, v12.x, v14.x etc. This means that they expose a matured, long term supported and backward

compatible set of APIs. Generally, LTS versions are actively supported for a year and enter into maintenance mode for another 18 months. A detailed release cadence and support process related information is documented here: **https://nodejs.org/en/about/releases/**

As of writing this book, we will go with v14.x as it is in active LTS phase. You can install Node.js from **https://nodejs.org/en/** in MacOS. The default configurations will be sufficient in most cases.

The following figure is the screenshot of an introductory dialog window from the installer:

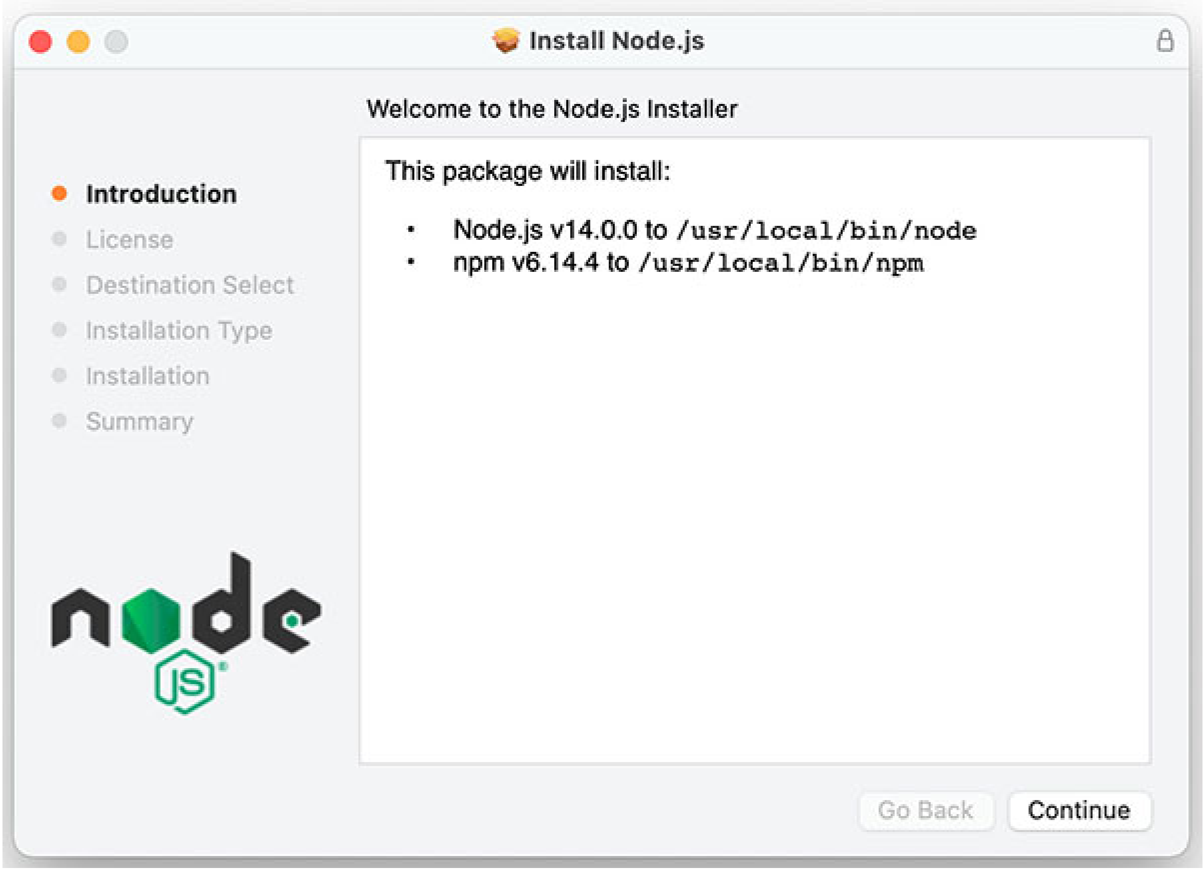

Figure 2.1: *Node.js installation step 1: Introduction*

The following figure is the screenshot that shows the next step, license agreement:

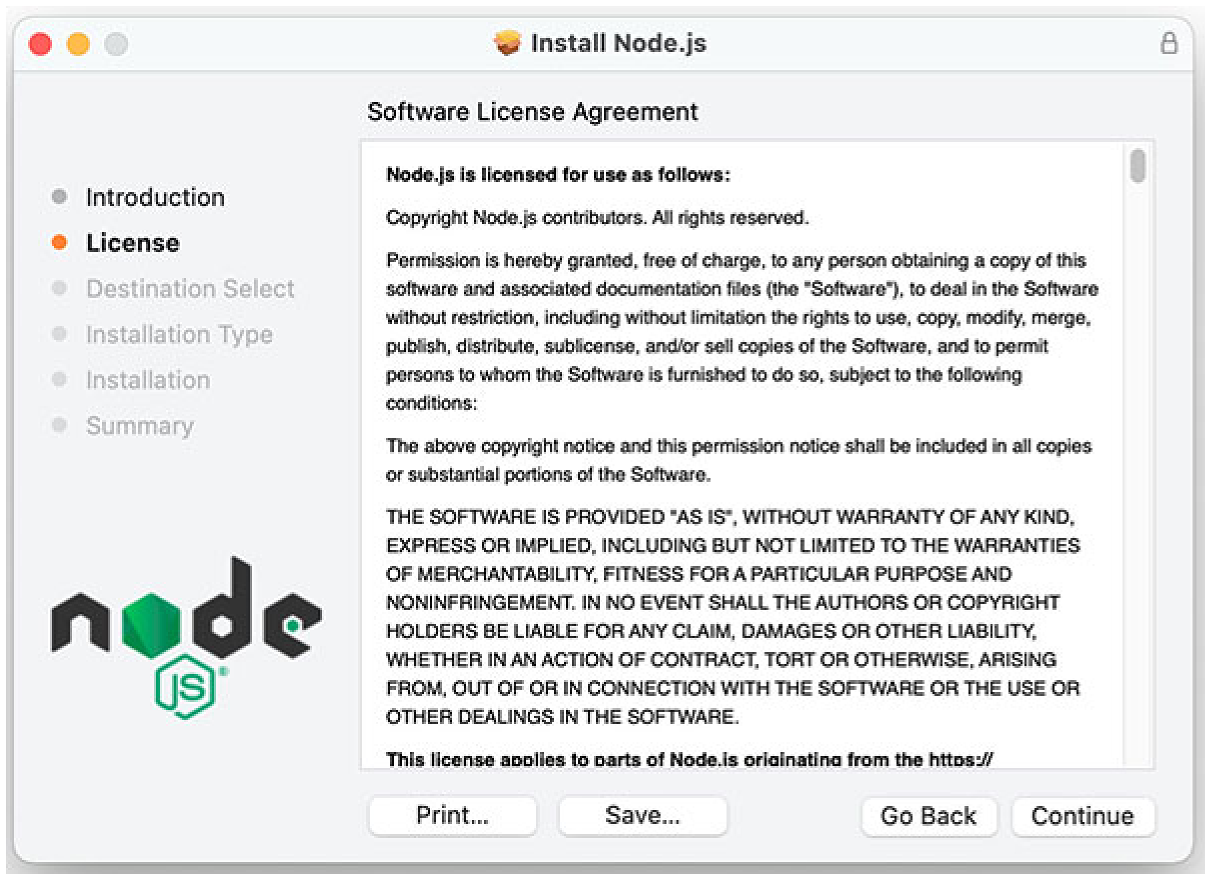

Figure 2.2: *Node.js installation step 2: License*

The following figure is the screenshot of the license agreement dialog window:

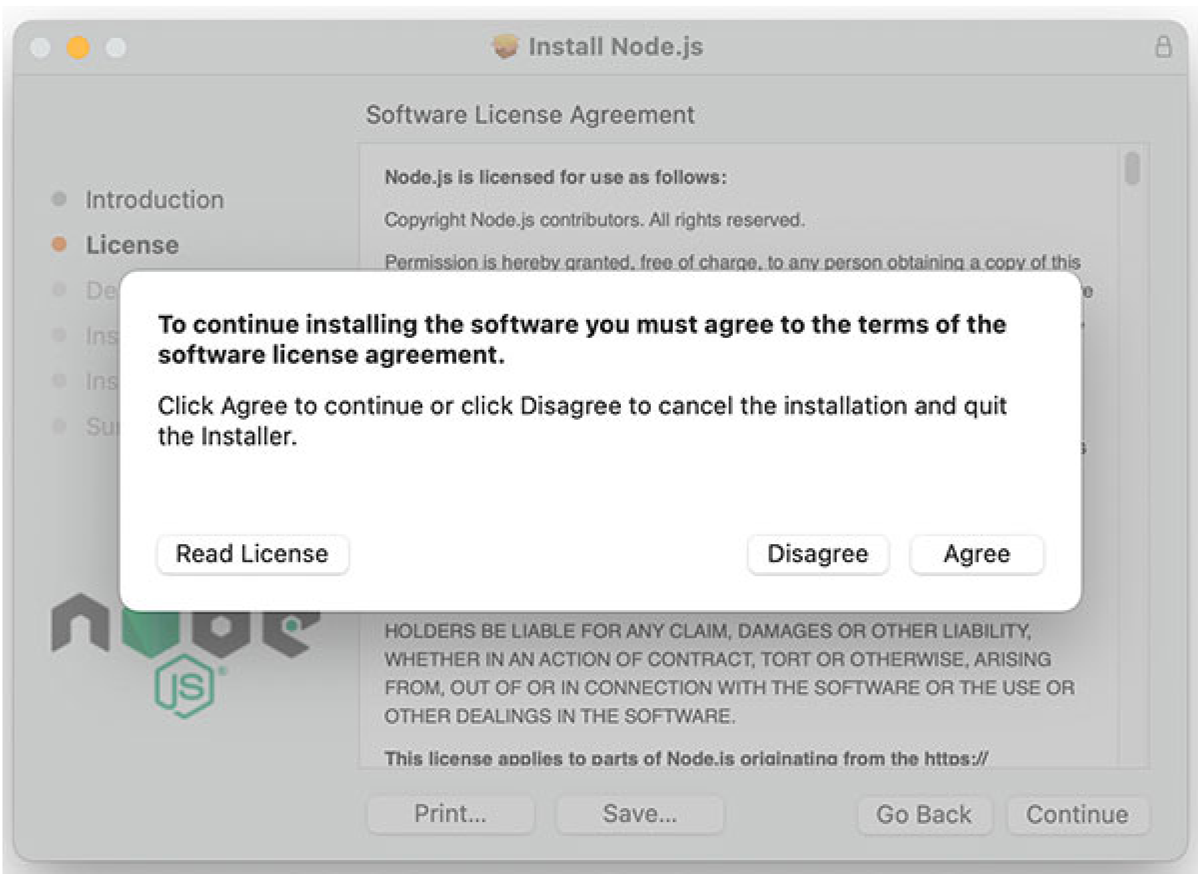

Figure 2.3: *Node.js installation step 3: License Acceptance*

Next, the installer examines the file system for the installation and validates the disk space, as follows:

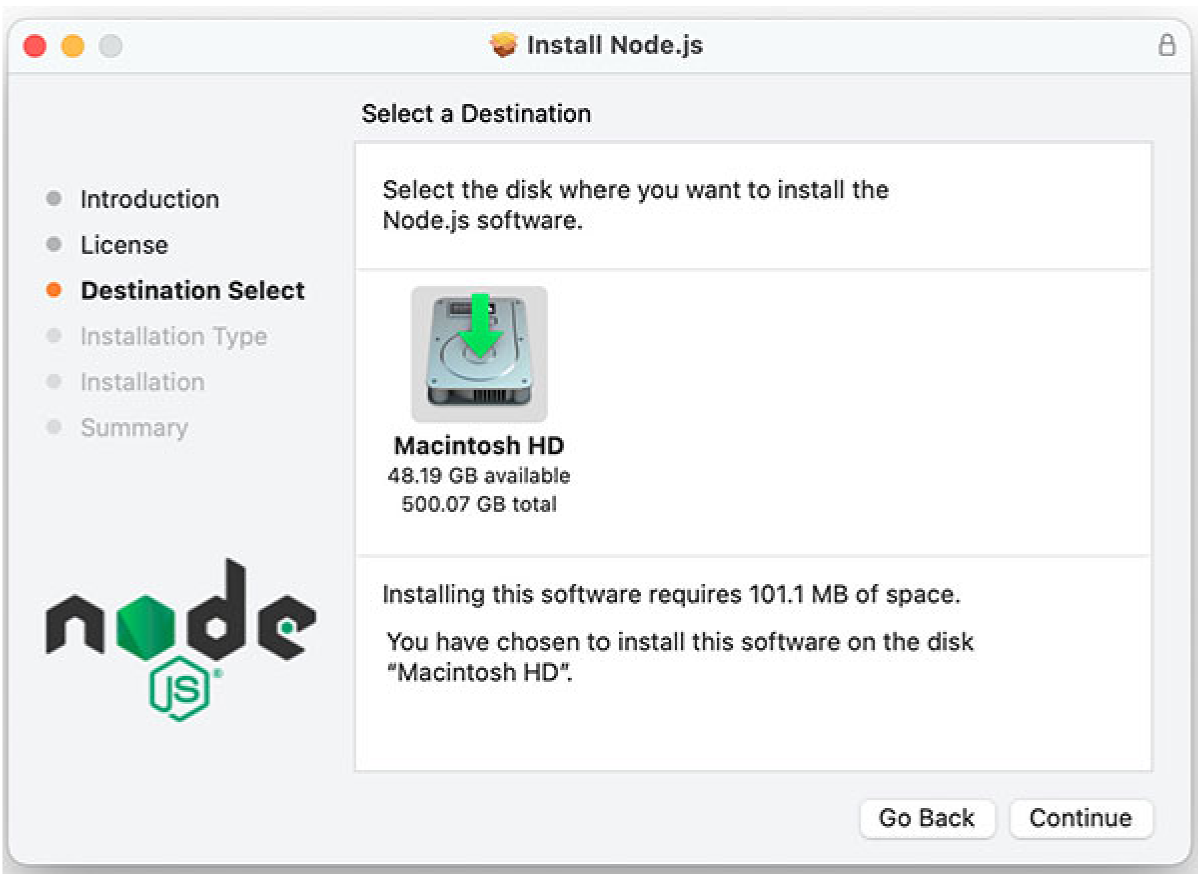

Figure 2.4: *Node.js installation step 4: Destination select*

Subsequently, the installer allows you to select a destination location, as shown. It also has additional customization options on the installation:

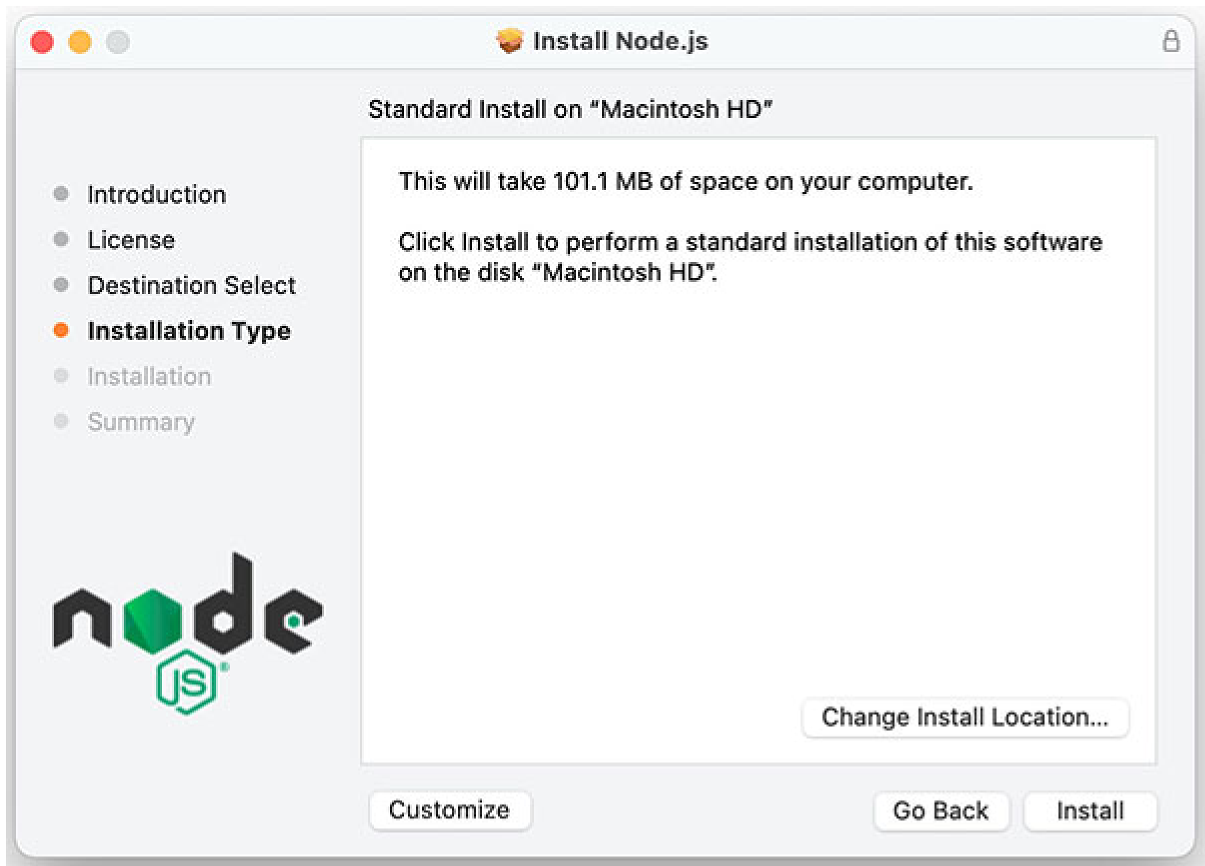

Figure 2.5: *Node.js installation step 5: Installation type*

Once all the user input has been received, the installer starts the installation, as follows:

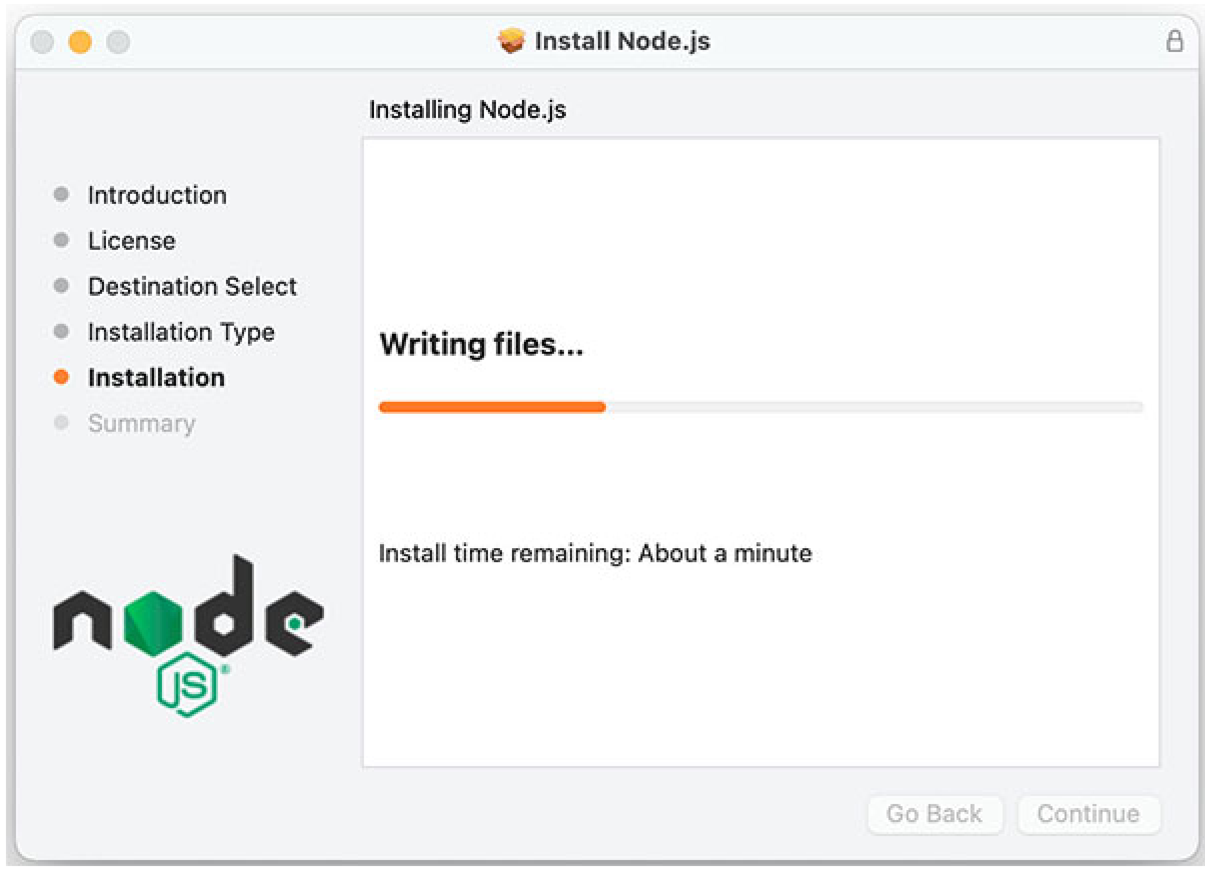

Figure 2.6: *Node.js installation step 6: Installation start*

The following figure is a screenshot of the installation progress:

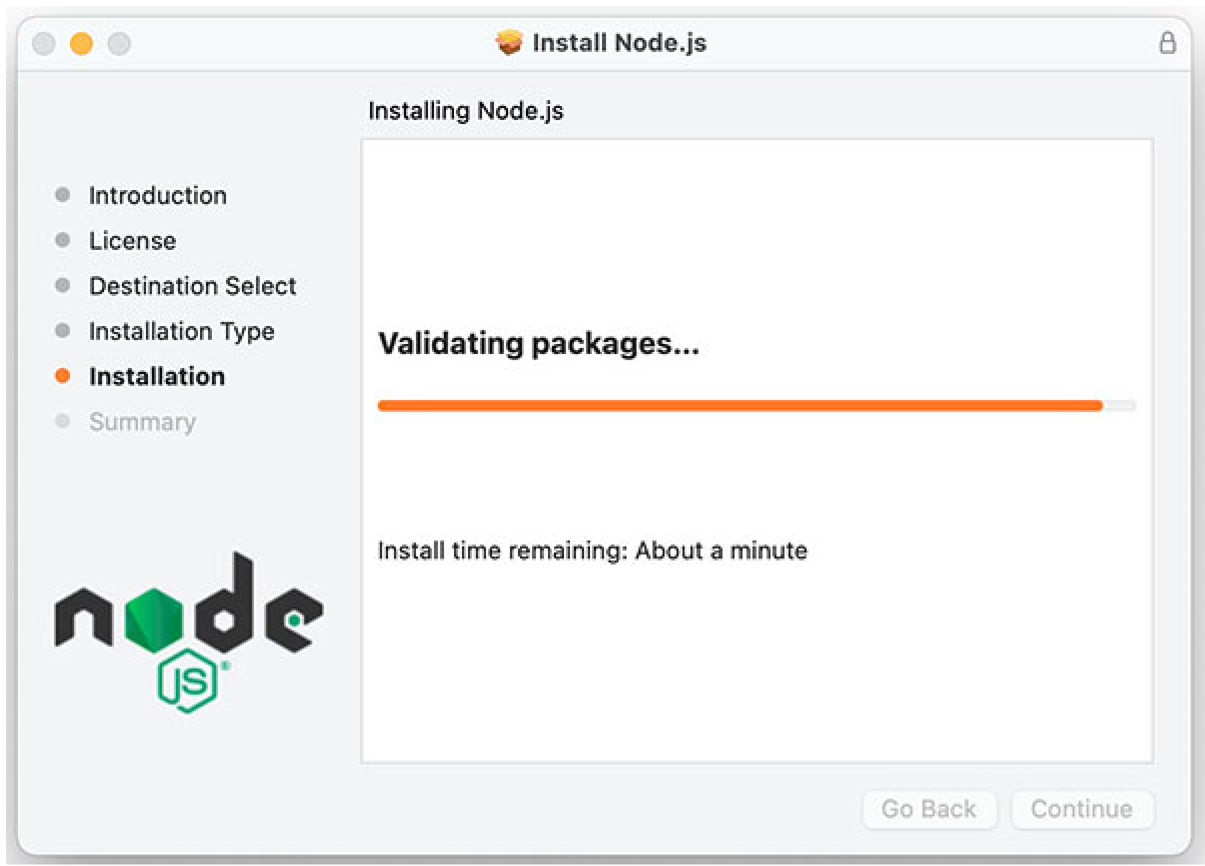

Figure 2.7: *Node.js installation step 7: Installation progress*

And finally, the installer shows the summary of the installation, as follows:

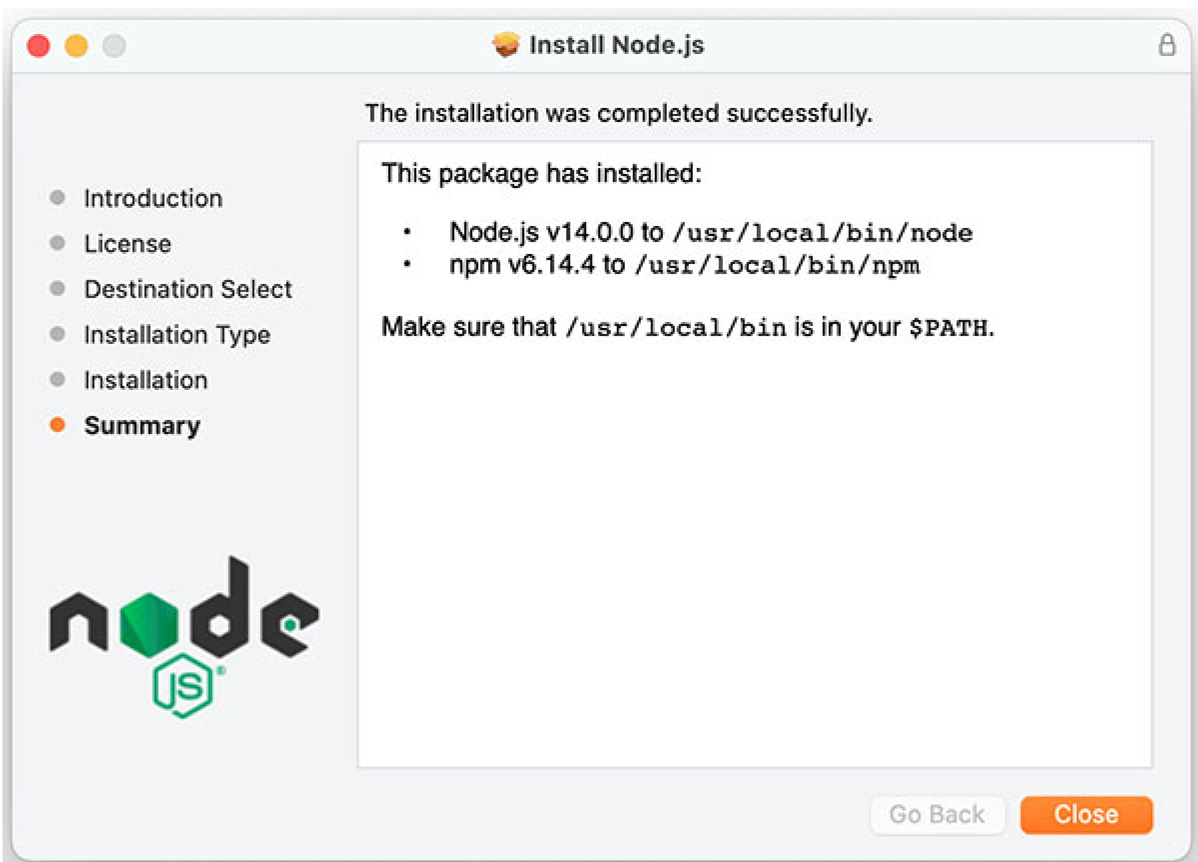

Figure 2.8: *Node.js installation step 8: Installation complete*

And at the end of it, we can confirm the installation and version as follows:

Microsoft Windows:

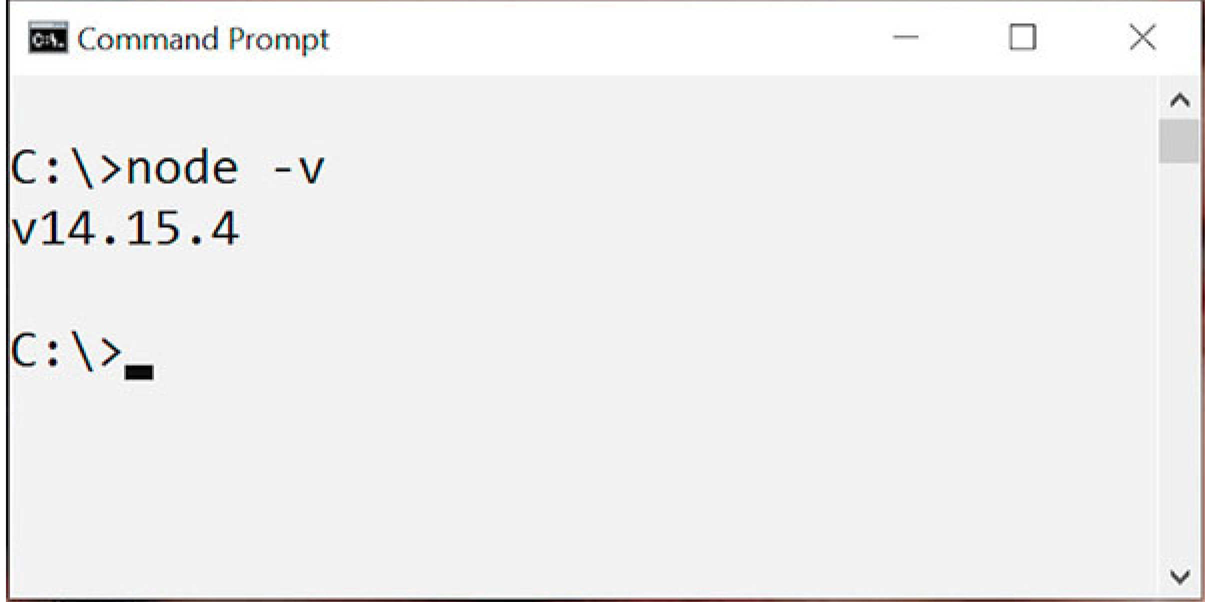

Figure 2.9: *Node.js installation and version verification in Windows*

Linux and MacOS:

```
2 — -bash — 45×8 — ⌘2
#node -v
v14.15.4
#
```

Figure 2.10: *Node.js installation and version verification in UNIX*

Dependencies selection

As we recollect from the title of this book, our objective is to develop a web application from scratch by making use of only core Node.js APIs. This means we don't have any dependencies with third-party node modules.

Resource requirements

While we will need to meet large-scale system requirements for hosting our server in production, these can be simple and trivial for the development purposes. Some minimum expectations are as follows:

- **Memory:** 8 GB
- **Disk:** 2 GB
- **CPU:** 2GHz frequency

> **Note:** One of the main success factors for Node.js is its very low resource requirements. Being single executable binary, Node.js is preferred runtime for many platforms where application stack is bootstrapped through runtime orchestrations.

Code editor selection

There is no static assumption made in this book about any specific editors, and we are not exercising any editor features as well. You can use your favorite editors for developing the application. The preferred option is 'vi' editor.

Any JavaScript editor works well for Node.js development. The common JavaScript editors/IDEs are as follows. A key advantage of using a sophisticated, special purpose editor is the syntax highlighting, not to mention a number of other cool features.

- 'vi' editor (comes by default with UNIX systems)
- 'notepad' (comes by default with Windows systems)
- Visual Studio Code: **https://code/visualstudio.com**
- Sublime Text: **https://www.sublimetext.com**
- Atom: **https://atom.io**

Conclusion

In this chapter, we learned about a number of aspects about the execution environment and system requirements for using Node.js for application development. This includes the supported and recommended platforms for Node.js, Node.js version selection and LTS philosophy, resource requirements, and code editor considerations. In the next chapter, we will examine some fundamental considerations of a website developer, such as architecture, functional features, performance, security, and debuggability. Learning these will empower us to write code that can be used in a real commercial setting and sustained for longer.

CHAPTER 3
Introduction to Web Server

In this chapter, we will formally introduce the concept of web server and define its core components. Further, we will examine the fundamental considerations and concerns of a website developer. We will determine what makes up a web server and understand the most common best practices that can be followed while developing an efficient web server. Much of these considerations will be carried over, discussed in detail, and leveraged in the subsequent chapters, when we deal with the specific components of our web application. For that matter, we will introduce the concept here for completeness and move on.

Structure

In this chapter, we will cover the following topics:

- Introduction to web server
- Web server basic components
- Concerns and considerations of web servers
- Web server considerations: Architecture
- Web server considerations: Performance
- Web server considerations: Security
- Web server considerations: Reliability
- Web server considerations: Extensibility
- Web server considerations: Maintainability
- Web server considerations: Serviceability
- Web server considerations: Observability

Objective

After studying this chapter, you will be able to understand the Node.js philosophy around event-driven architecture with asynchronous programming. While meeting that objective, you will also understand workload efficiency-related concepts, the performance characteristics of web workloads, and various tradeoffs that exist in resource usage versus performance. You will also learn about some of the core Node.js features that are relevant to our goal of developing a web application, and understand the basic definition of a modern web server application and its internal composition. Further, you will understand the main considerations when building a web server, with their effects on specific capabilities of the said web server. More specifically, you will understand how principles of software engineering apply to a web server: architecture, performance, security, reliability, extensibility, maintainability, serviceability, and observability. This learning will also be useful when we deal with individual components while building our web server application.

Introduction to web server

A web server is a software that serves web content to its clients over a network. The content can be any one or more or a combination of predefined and hand-written static pages or dynamically crafted HTML pages, or it can be multimedia content obtained from another software. Based on the specific use case that the server handles, the server may be performing a number of actions within its scope of execution. The produced web content may be seen as a manifestation or final product of those actions. The following figure illustrates the simplest form of a web server:

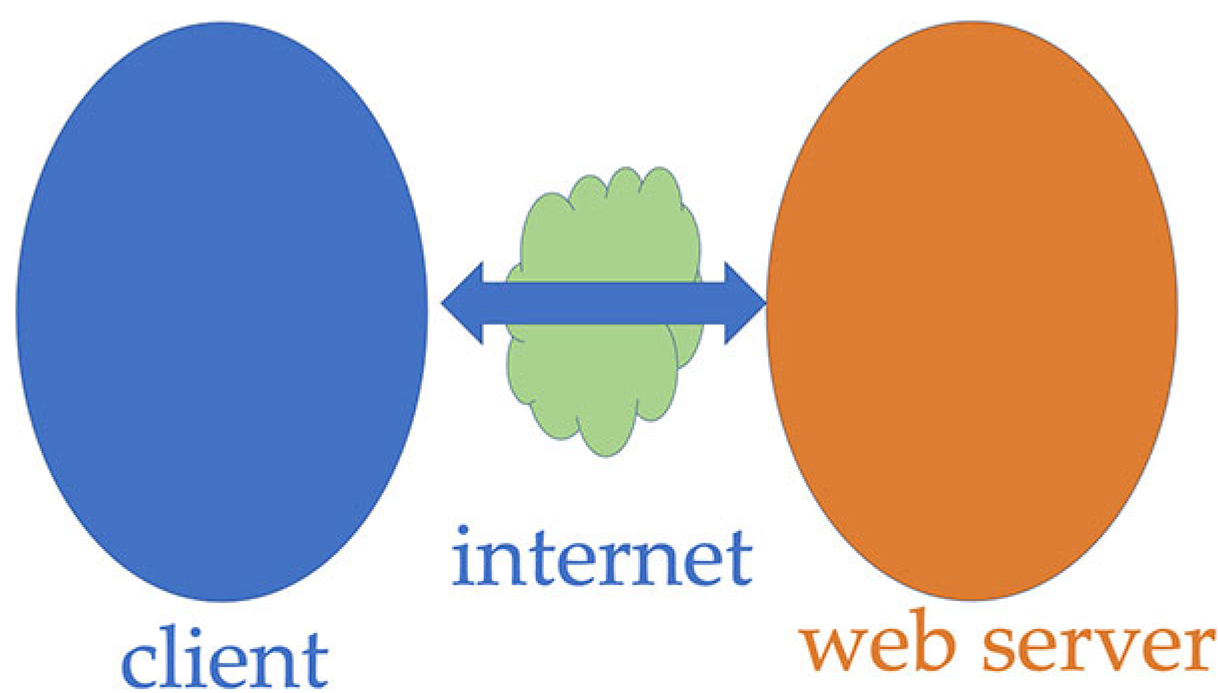

Figure 3.1: *A simple web server architecture*

Web servers represent an important phase in the evolution of software engineering. A web server represents heavily reusable software without needing a license fee to use it. Hosting the software at a central location and making it accessible through a public network with a well-defined protocol improves its reachability and reusability by large.

Core components

While there is no easy way to generalize hundreds (if not thousands) of web server architectures that are in use today, the componentry of a web server is depicted here:

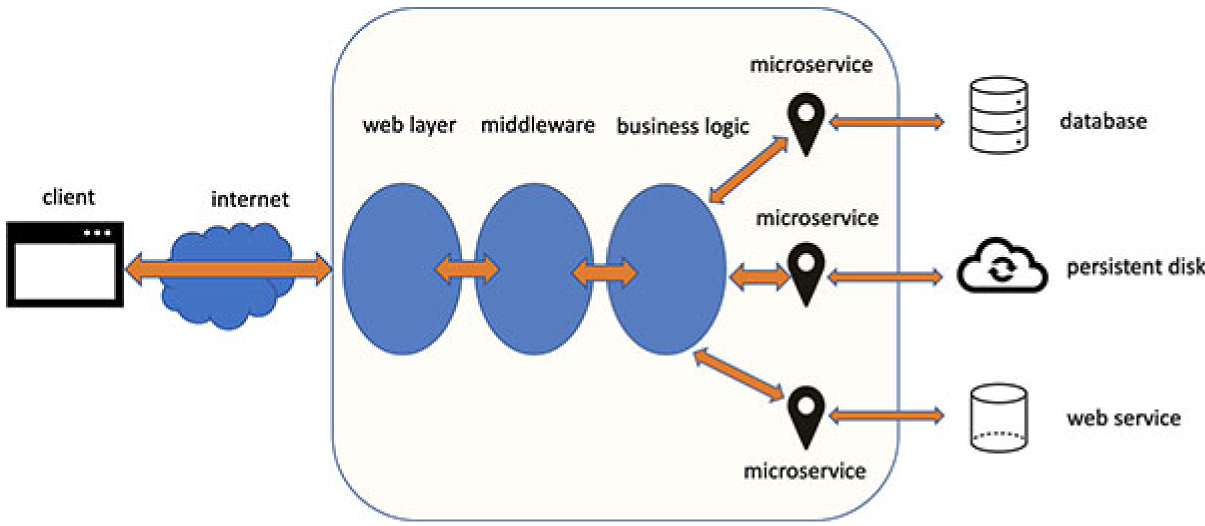

Figure 3.2: *A practical web server architecture*

Client

A client is an entity that consumes the services of a web server. Typically, this is a web browser, but it can also be command line tools or any custom program that understands the protocol of communicating with a web server.

The communication happens over the internet, in the most common use case.

Web layer

This acts as a proxy for the backend components. The web layer intercepts the requests from the clients. Also, it composes the response into a presentable form for the client.

The advantage of having this layer is the ability to abstract heterogeneous types of clients—desktop browsers, mobile devices, sensors, testing tools, and so on—can seamlessly interact with the web server, and the varying communication styles for the different endpoints do not need to live beyond the web layer.

Middleware

Middleware in general sense is a software component that interfaces various entities of a distributed computing system. In the context of a web server, a middleware is simply the request–response pipeline. That is, all the

processing on the client request and the server response are managed in the middleware. The request data is parsed and extracted out of the request object before passing it to its backend components. Similarly, the response data is composed into the response object in the middleware on the return path of the request–response pipeline.

Tip: The key functions of web middleware are: managing request data, implementing session, and managing protocol headers.

The advantage of middleware component is clear isolation between the business logic layers and the web server layers; in other words, it is the separation of concerns. When our web server becomes the subject of debugging, bug fixes, extensions, enhancements, and re-architecture, this modularity comes in handy as the developer can easily identify and focus on the correct code.

Business logic layer

This layer implements the core business logic of the web server. For example, if we are building a flight booking application, the business logic layer would implement:

i. Reservation/cancellation feature
ii. Flight schedule view feature
iii. A customer profile management system at the least

The business layer in this case would act as a composition of three capabilities, and those three capabilities are delegated to individual microservices placed next in the architecture diagram. In yet another example of a natural language processing application, the business logic layer would implement the following:

i. A data receiving and preparing feature
ii. A language processing pipeline feature
iii. A data aggregation and presentation feature

The business layer would simply act as a container for these discrete capabilities while the actual capabilities are implemented in individual sub-components or crafted into a micro-service callable from the business layer.

The advantage of the business layer is trivially obvious: it contains the core logic pertinent to the business of the web server and are subject to changing demands from its users and customers. The developer is able to enhance the logic without worrying about other parts of the server functions.

The microservice layer

This layer specializes in a specific aspect of the business logic. In the above-mentioned example of flight booking, the microservices implement reservation or cancellation, schedule view, and customer profile management function. To implement the said functions, these services may be leveraging capabilities from services further downstream in the architecture or from remote web services accessed through the Internet.

The advantages of business layer apply as is to microservices, as these are specialized ingredients of the business layer itself. Additionally, lifecycles of microservices can be independently managed. This means if we find that a specific microservice is a performance bottleneck in the entire application, we can scale it up to share the load and relax the bottleneck. This is because the service is loosely coupled with the main trunk of the application, as the communication is over the network as opposed to a direct function invocation.

In summary, microservice architecture provides a high degree of flexibility to the application. While acting as a vital ingredient of the application, it is inherently decoupled from it, leading to independent lifecycle management.

Note: We defined web server as a software that serves web content to its clients. Mainstream use cases of a web server also include received content from clients. Think about filling a form, uploading a file, etc. These use cases do not change any of the premises, architecture, or design of the web server; it instead just reverses the data flow in its request-response pipeline.

Concerns and considerations of web servers

Understand the problems inherent in the web server architecture and identify best practices around those issues to properly scope the placement of web servers in the spectrum of software architecture. Let's compare a

web server software with a simple desktop application that provides a trivial software capability, such as image processing (say user supplied images are color-inversed).

Comparison of a desktop and a web server

The two systems are symbolically represented as follows:

Figure 3.3: *A desktop application and a web server*

Similarities

- Both the desktop app and the web app are driven by user (client) actions
- Both the desktop app and the web app perform computation
- Both the desktop app and the web app produce result

Differences

- A desktop app is installed individually on the client's system, whereas a web app is installed at a central location (such as a server hardware).
- A desktop app may be subjected to the vendor's license fee, while web apps are not.
- In a desktop app, the user actions are directly translated to events that trigger processing actions, most probably direct function calls. On the other hand, in a web app the user actions are first translated to a request, which is wrapped under a protocol and then sent across the network to the server. In the server, the reverse action takes place,

wherein the protocol is terminated and the request is parsed and processed.

- In a desktop app, the computational resources consumed for the processing has bearing on the user interface and vice versa. In other words, both the user interface and the backend image processing reside and run in the same operating system process. In a web app, these are fully decoupled - the user interface and the request processing actions are performed in different processes and computer systems with potentially different operating systems. This trivial-looking difference has visible implications to the web server architecture and various issues that we will examine later, but contemplate over this.
- A desktop app is usable by a single user at a time, while a web app is inherently multi-user. This means a single web server process may be handling multiple clients simultaneously–remember the topic of concurrency that we learned in the first chapter.
- In a desktop app, there is near-zero latency for a processed request to be rendered in the view, while in a web app, this depends on few factors such as the speed of the network, the number of concurrent connections, the number of server instance replications, and so on.
- A desktop app is secure from external interferences, while a web app is vulnerable against various threats, as there is a public medium between the two layers (client and server).
- A desktop app becomes less reliable if the hosting system crashes. A web app can be made highly reliable by making provisions for fast recovery, including replications.
- A desktop app's state represents the current user's session. A web app's state represents the session of all the currently connected users. This means, to debug application issues, a web app will need special logging mechanisms to customize the runtime state as well as the application state to a specific user's context.

There are more differences that can be drawn – discrete or derived, but we will stop here for now. The main intention of this comparison is to dissect the two software architecturally and showcase how different they are, with various tradeoffs. This differentiation will take us directly to the need of

discussing each of these differences in detail with pros and cons. And focusing on the drawbacks of the web server architecture over the desktop application will lead us directly to the best practices while developing server software, and those best practices will mostly make up the rest of this book.

Note: It is surprising to observe the degree of divergence occurring to software components when their architecture changes even slightly. Among other things, it emphasizes on the critical importance of the role of architecture and why we should be spending enough time on the architecture while developing a software.

Web server considerations: Architecture

At the core of a web server is a server-side entity that serves web content. The simplest representation of a server is as shown in the following figure:

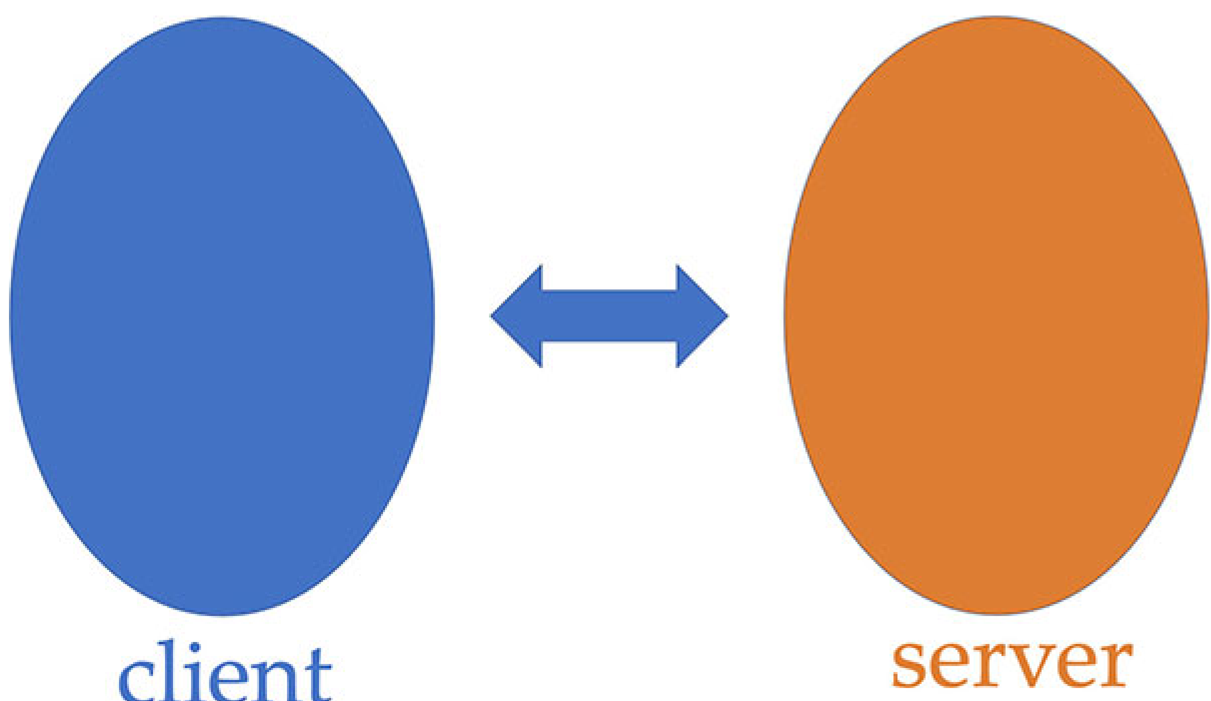

Figure 3.4: *A simple client-server architecture*

In the preceding picture, the client can be a browser, a crawler, or any software that adheres to a network protocol to talk to the server. However, for a durable client-server experience, the server needs to qualify in certain

aspects so that the peculiarities discussed in the previous section do not cause any issue to the web server.

Architecture, in general, is a set of components and their interactions. The most common web app architectures are client–server, peer–to–peer, and microservice architectures.

In the above diagram, the server is a single entity encapsulating a lot of capabilities. This is fine for visualization, but in the real world, a single program dealing with a number of things will neither function nor scale properly under various circumstances.

Microservice architecture

An obvious solution to this is to split the program into modules, with each module encapsulating a discrete function of the web server. This is a good first step and improves readability and maintenance, but it does not fully address the reliability and scalability issues.

At this point, we consider microservice architecture as the preferred choice. This addresses the readability and maintenance issues and also takes care of scalability issues.

But it also brings its own issues: performance as well as complexity of deployment and monitoring. For example, running a command in a command line or a simple script was sufficient for deploying a single program, but now we need to execute a number of scripts in a pipeline to get all the microservices up and running, while properly dealing with the complexities involved when one of the services does not come up. Should we retry or rollback? If retrying, how many times should we do so before declaring a failure? How show we interpret the log messages emanating from different services? How can we produce an aggregate view of the application?

These are problems that also have known solutions. Additionally, these issues are empirically proven to be less pervasive as compared to the earlier issues of scalability and maintainability, so our preferred architecture for the web server is microservice architecture.

Here’s a simple representation of microservice architecture:

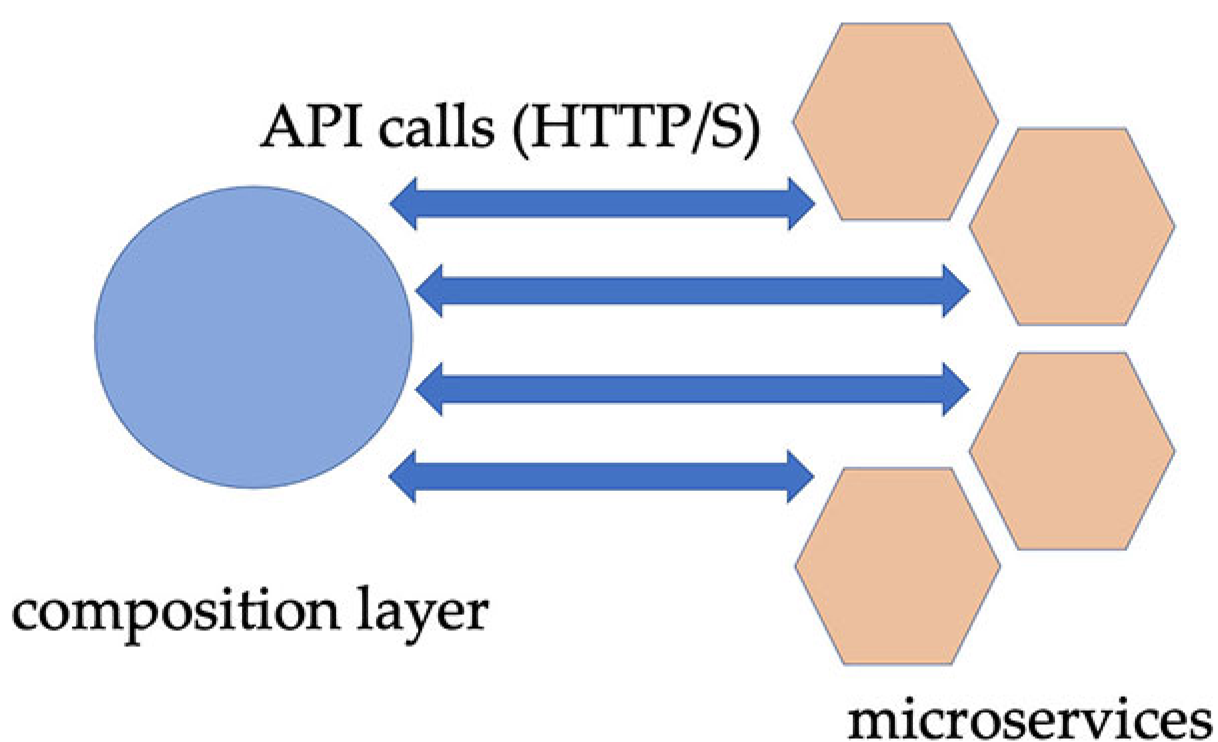

Figure 3.5: *A typical microservice architecture*

Web server considerations: Performance

At the beginning of the chapter, we compared a web app with a desktop app and made an observation that the former will perform slow as compared to the latter by virtue of the proximity of the request processing entity (the server or the backend) and the rendering entity (the view). In the desktop app, the view communicates directly with the backend, in-process through a method invocation.

This is illustrated in the following figure:

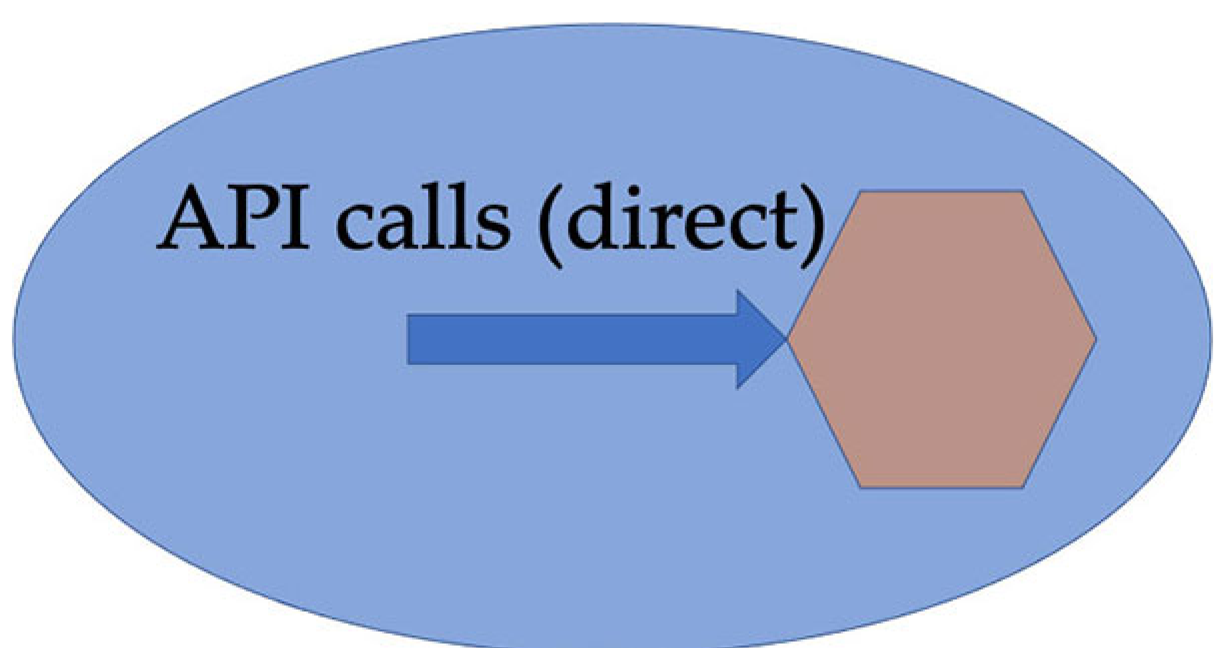

Figure 3.6: *A desktop application where the calls are direct*

In a web app, this communication is made over the network, with TCP, HTTP and/ or other protocol overheads. The overhead persists all the time and occurs four times in one communication:

1. Overhead of building the protocol header on the client side when the request is made.
2. Overhead of unwrapping the header on the server side when the request is parsed.
3. Overhead of building the protocol header of the response on the server side before it is sent to the client.
4. Overhead of unwrapping the header in the response at the client side.

This is illustrated in the following diagram with the hotspots of performance degradation highlighted in red circles.

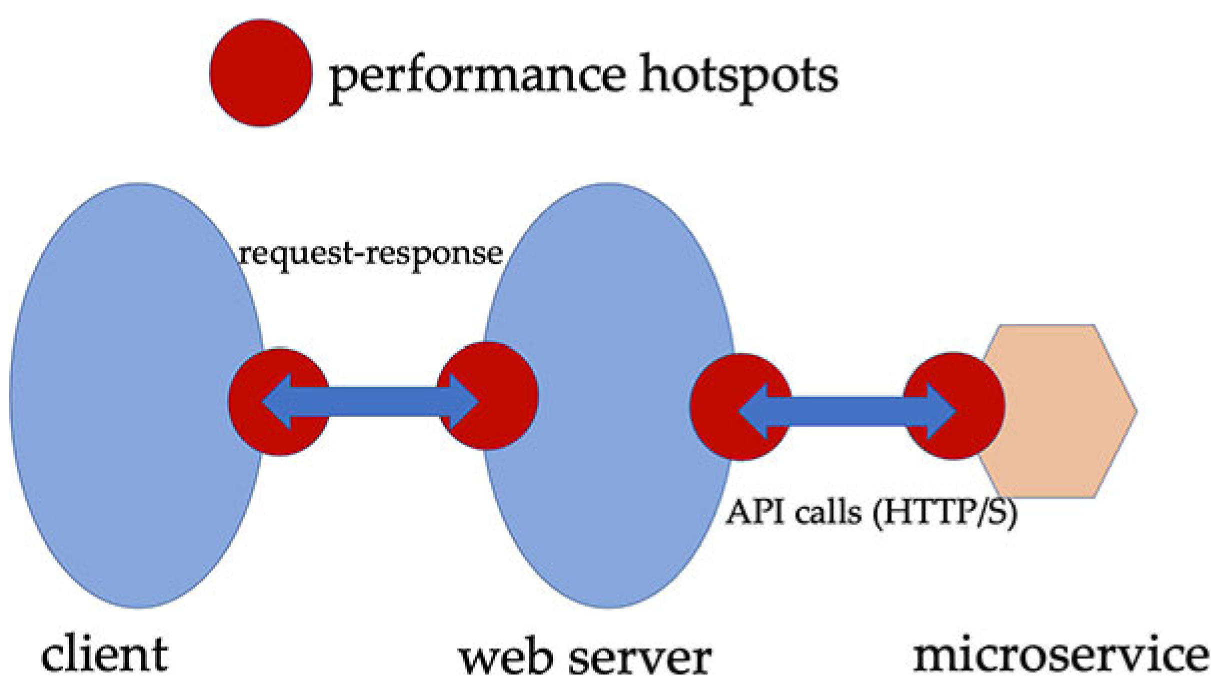

Figure 3.7: *A web server application where the calls are indirect*

So inherently, a web app will be slower than a desktop app.

In addition, further layers in the web server (brought for modularity and independent lifecycle management) each bring the same proportionate performance overhead as the overhead of the communication between the client and the server.

Performance overheads

In a nutshell, here are the main inhibitors to performance:

1. The overhead of network transport (request and response)
2. The overhead to packing and terminating protocol headers (network, application layer)
3. The overhead of processing (understanding) the request
4. The overhead of composing (formulating) the response
5. The overhead of switching between concurrent clients
6. The overhead of switching between different tasks of a single request response cycle

The net result is that the end user experience grows weak with latency (the time between the request issuance and the response rendition).

This means a web application should strive to improve its response time, wherever possible.

The first item in the list of factors is a natural side effect of being a web server. The best thing to do to improve this is network performance tuning at the operating system.

The second item is partly a natural side effect of being a web server, and partly due to the selection of an application-level protocol such as HTTP. When we add security to this, the protocol changes to HTTPS, which adds more overhead to the performance.

Under normal deployment circumstances, there is nothing we can do to change this. If you are deploying under a container orchestration system with a cluster and a load balancer, there are techniques to terminate SSL at the load balancer level etc. But as of now, we are good with HTTP and HTTPS as is.

Overhead of processing the request and composing the response is not new; those overheads are at par with the overhead of a desktop application.

Overhead of switching between clients is a side effect of Node.js. As we have only one thread to run the application, a single thread needs to switch between all the connections.

Best practices

A best practice is to ensure that the core business logic (code between the request and response) does not carry large and complex operations such as deep loops, long processing of binary data, and so on.

Overhead of switching between different tasks of a single request response cycle is also a side effect of Node.js. This switching is managed within the Node.js core platform, so we don’t need to worry too much here.

In summary, given the architecture and design considerations, the most natural best practice around performance of a web server is to avoid unnecessary I/O and minimize the CPU bound operations between its request and response phases.

Note: You may recollect the objective of Node.js from the first chapter: Given the nature of the workload in the web application, how can we best separate and multiplex the CPU bound and I/O bound work, while transforming all the blocking code to non-blocking and defining convenient language-level semantics to manage the asynchronous context: ultimately to improve the performance of web workloads.

Web server considerations: Security

Given that our web server is centrally placed and accessed over the network, the server software is subject to a number of security threats. In fact, web server is the software architecture that is the target for most security exploits. The threats include:

- Tampering with the privacy and integrity of the server software
- Tampering with the privacy and integrity of the data that the server deals with
- Tampering with the privacy and integrity of the resources that the server deals with
- Tampering with the privacy and integrity of the transport channel that the server uses to communicate with its clients

In short, all the security threats can be generalized into one model: tampering with the privacy and integrity of data and resource at endpoints that the server has access to. This is clearly a side effect of the network-based architecture that the web server uses but it is something we are accepting as a tradeoff for improved code reusability, stability, and user experience. In addition, the good thing with security is the fact that most of these issues can be addressed at the server side itself, which means these concerns can be concealed from the user who accesses the web server through a browser. So, they are oblivious to many of these concerns.

Security threat types

The following figure shows the top 10 web server security threats:

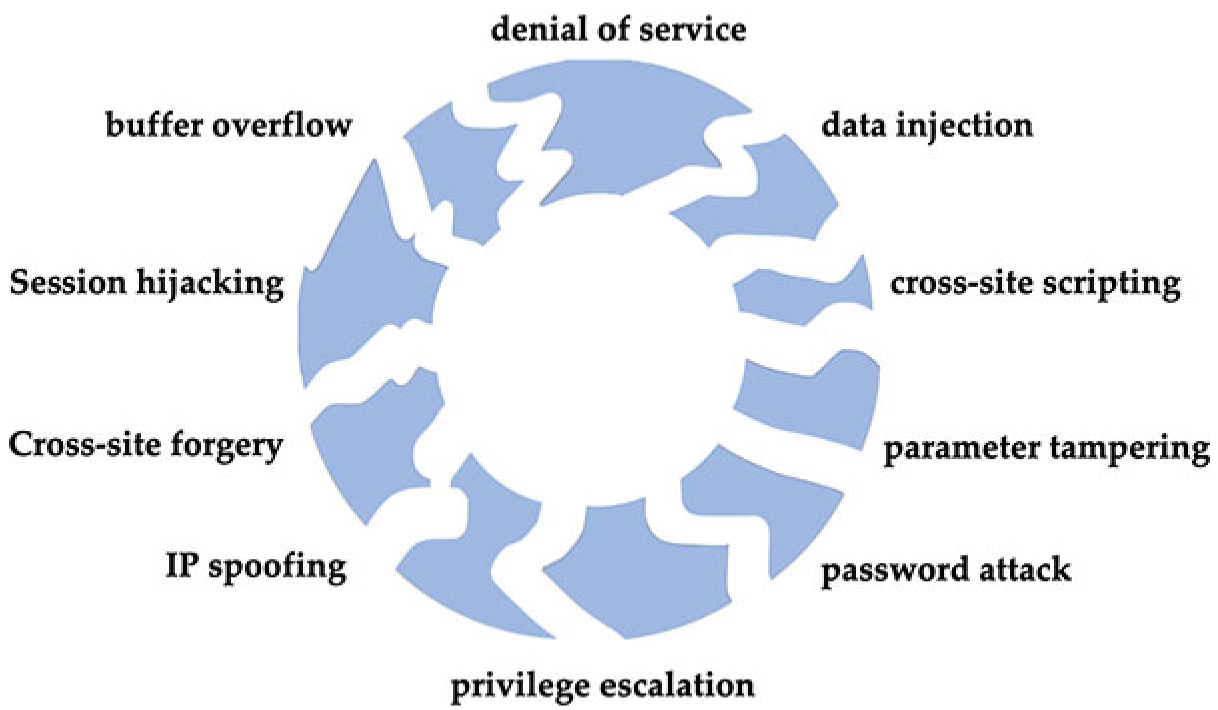

Figure 3.8: *A graphical illustration of top web server security threats*

Tip: Cyber security is a vast topic and concerns a number of security themes. Lot of research happens in this area, and it is also a specialization paper for many universities.

Best practices

Here are some of the common best practices around web server security to alleviate the above-mentioned issues:

- Understand what portion of code and data are subject to privacy and integrity
- Define the scope/span of code and data that your web server accesses/processes
- Define the scope/span of resources that the web server accesses
- Understand the scope/span of transport channel the server communicates through
- Use authentication mechanism to protect the code, data, and resources
- Define roles (authorization) to implement the said protection hierarchically

- Treat every data-intake point as untrusted and ensure proper validation
- Parse and prune scripts, commands, and queries that originate from the user
- Treat all data that flows over the network as exposed and ensure encryption

In addition to this, ensure that the entire application stack is up to date with the latest security fixes. Also, constantly auditing the application to check whether all the usage of the web server is genuine and detect outlier patterns and address them in a recurring manner.

These are the most common best practices for our web server. It is surprising to see the cost of securing a software that is centrally placed in the internet! However, we prefer the web-based approach over a desktop program as the pros outweigh the cons.

Web server considerations: Reliability

Reliability for a web server generally refers to its ability to be available and consistent with respect to the external interface and its function. From a client's perspective, a reliable server is considered available at any time. Also, a consistent response is obtained for a specific type of request sent.

In a narrow sense, availability is different from reliability, but we will treat availability as a subset of reliability for our discussion.

At the enterprise deployment level, there are several techniques to ensure high availability of the server, including clustering, load balancing, disaster recovery, and so on. However, we will not talk about those techniques here as our baseline is a simple desktop application; instead, we will focus on the single instance of our server.

Examples of reliability issues

The common types of reliability issues are as follows:

- Server crash
- Server unresponsive/hang
- Operating system crash

- Hardware failure
- Traffic overload

Crash is a situation when the server process terminates abnormally when it is not expected to. The reasons are bad code, bad data, or both. Crash leads to broken client requests as well as call drops, resulting in unavailability of the server until it is booted up again.

The following figure illustrates an example of a crashed process:

```
node — root@d1: ~ — -bash — 44×7 — ⌘2
#./node app.js
Segmentation fault: 11
#
```

Figure 3.9: *An example of a crashed process*

Hang is a situation where the server process is alive but the thread is unable to progress and handle the work when it is expected to make progress. The reason is usually bad logic in code, due to which it waits for a non-existent event occur. Implication of a hang is the same as that of crash–the current requests are abandoned, and the server is unavailable until it is restarted.

The following figure shows an idle process that consumes 0 CPU for 10 hours or so:

```
node — root@d1: ~ — bash — 58×10 — ⌘2
#top -a
ID     COMMAND %CPU TIME       #TH    #WQ   #PORTS MEM     PURG
12023  node    0.0  10:07.00   6      0     27     6164M   0B
#
```

Figure 3.10: *An example of a hanging process*

Operating system crash occurs when a bug in the kernel manifests. The reason is trivial. As the system reboots upon an OS crash, the implication to the clients is the same.

Hardware failure may imply a long delay to clients, as the recovery from the failure may not always be realized through a reboot.

Traffic overload is a situation wherein the resources in the server (memory, CPU, disk, ports, etc.) are exhausted or nearing exhaustion. The reason is more concurrent clients than normal. The implication is delay in addressing client requests, eventual crash due to exhaustion, or both.

Best practices

Common reliability best practices are as follows:

- Define key reliability metrics (**Service Level Agreements** (**SLAs**) and **Service Level Objective** (**SLOs**)). This helps identify automated mitigation plan upon unavailable/unreliable server.
- Loose coupling between components. This helps isolate and refresh faulty components without bringing everything down.
- Automation of deployment. This helps in faster bootstrapping.
- Minimize startup time. This helps in faster bootstrapping. Node.js has a specified objective to be faster on startup time, and it is one of the fastest bootstrapping runtime at present.

- Be stateless (no local persistence). This helps in reconciling broken states from safe and stable endpoints rather than fixing from a broken file.

Note Service Level Agreement and Service Level Objective together refer to a set of predefined terms of understanding between the producer and the consumer of a software service in terms of the expectation on quality, responsiveness, and cost of the software as well as the responsibilities of both the stakeholders. This agreement acts as a governing principle for the producer and consumer in all phases of development, delivery, and production.

Web server considerations: Extensibility

Extensibility is the measure of making amendments to the server program after it has been developed and deployed. Software is known to evolve over time due to the continuous feedback that it usually obtains from the field (userbase). Based on the original architecture and design, these evolutionary changes can affect every part of it, including the architecture itself – the highest level of abstraction of the software.

Examples of extensibility

Some examples of future growth for a web server are as follows:

1. A custom middleware function that every request passes through. In this example extension, the new feature introduces a new logging routine or a new validation routine that touches every client request that reaches the server. The following pseudo-code shows a simple middleware that is invoked on every client request and logs a specific data element:

```
1. function customLogger(q, r, n) => {
2.   logger.log(q.query.name)
3.   n(q, r)
4. }
```

2. Extend the web server to support a new route (request type). In this example, the web server is able to handle a new request type. The request type may be characterized by its method, type of query data, or the type of response data that it expects from the server. The following pseudo-code shows a new route being inserted:

```
1. function handle(q, r) => {
2.   if(q.url === 'foo') {
3.     handleFoo(q, r)
4.   }
5.   // rest of the code
6. }
```

3. Support a new encryption algorithm. In this example, the web server is able to handle a new type of client, which wishes to communicate through a new data encryption algorithm that the server did not cover earlier. The following pseudo-code illustrates an example of leveraging a new cryptography algorithm 'AECS-256-FOO', which is implemented as an extension in the crypto API but the user is able to use seamlessly.

```
1. const c = require('crypto')
2. c.createCipher('AECS-256-FOO', 'foo')
```

In all the three examples, the key to extensibility is the ability to add the feature as a natural extension to the existing program, without needing to refactor the existing features to accommodate new ones.

Best practices

Here are some common best practices to achieve extensibility to our web server:

- You need to understand and document the following:
  - key assumptions made in the design. This helps in designing extensions and plugging them in extensions naturally into the existing design.

    - key data and data type definitions. This helps in fusing new types into the existing types, and/or leveraging existing types required for new extensions.
    - key feature functions. This helps in reusing existing code.
    - high-level code flow. This is very important to maintain the sanctity of the design and code after the server has been extended.
    - high-level data flow. This is very important, to maintain the sanctity of the design and data after the server has been extended.
- Define data types with provision for extensions. This means Classes and Data structures are designed as abstract where there is a potential for future extensions.
- Define capabilities with provision for extensions. This means functions are designed as abstract where there is a potential for future extensions.
- Parameterize key policies around code and data. This helps in implementing polymorphic functions that behave differently based on input. In other words, avoid implementing functions with assumptions on input data.
- Define architecture components loosely. This helps in re-architecting the server, if need be, without heavily refactoring the existing code.

Web server considerations: Maintainability

Maintainability of a web server refers to the ability to seamlessly maintain the software over a period of time. Maintenance generally refers to:

- Fixing bugs that are reported from the field
- Fixing security issues that are reported from the field
- Supporting newer platforms and environments
- Enhancing user guide based on user experience
- Enhancing code to improve performance
- Enhancing code to improve reliability

Importance of maintainability

All of these are relevant in the context of a web server. At the same time, it is worthwhile to note that there is no added burden on maintenance to the server by virtue of its architectural difference with our desktop application. In other words, all the above-mentioned items are applicable for a simple desktop application as well. In either case, the server software should aim to prepare itself for being subject to these maintenance activities.

Best practices

Here are some best practices around building highly maintainable software:

- Modularize the code around key abstractions and capabilities. This helps scope the subject of maintenance well within one or more modules.
- Ensure that the code is highly readable (by other developers). This helps isolate the subject of maintenance into one or more modules and understand the business logic just through code review. The design document or the user guide may not mention anything about the bug due to its nature, so code reading is the only alternative.
- Document key functions and interfaces. This helps scope the subject of maintenance well within one or more modules or APIs.
- Factor and modularize platform-specific code. This helps scope and isolate the subject of maintenance well within the relevant platform.
- Establish performance benchmarks with baselines. This helps in making meaningful comparisons and quantification of performance bottlenecks or degradations.
- Ensure that bug fixes do not ‘grow’ into features. This helps retain the software’s original architecture and design. Otherwise, the bug fixes can tamper the design objectives.
- Ensure that backward compatibility is maintained. This helps confine the features and bug fixes within a well-defined scope, instead of them escaping into regressions and cascaded issues.
- Define and follow semantic versioning. This helps organize maintenance artifacts of code in an easily traceable manner, with

proper audit trails. Also, classification of work items become easy, leading to a well-defined release process.

Web server considerations: Serviceability

Serviceability refers to installation, configuration, fault detection, fault data capture, problem isolation, problem recreation, problem determination, and applying fixes and patches, among other things. It is an important aspect of maintaining the delivered software, as the serviceability directly influences the warranty cost, which is the cost of supporting the software. The following is a graphical representation of an in-production software qualifier:

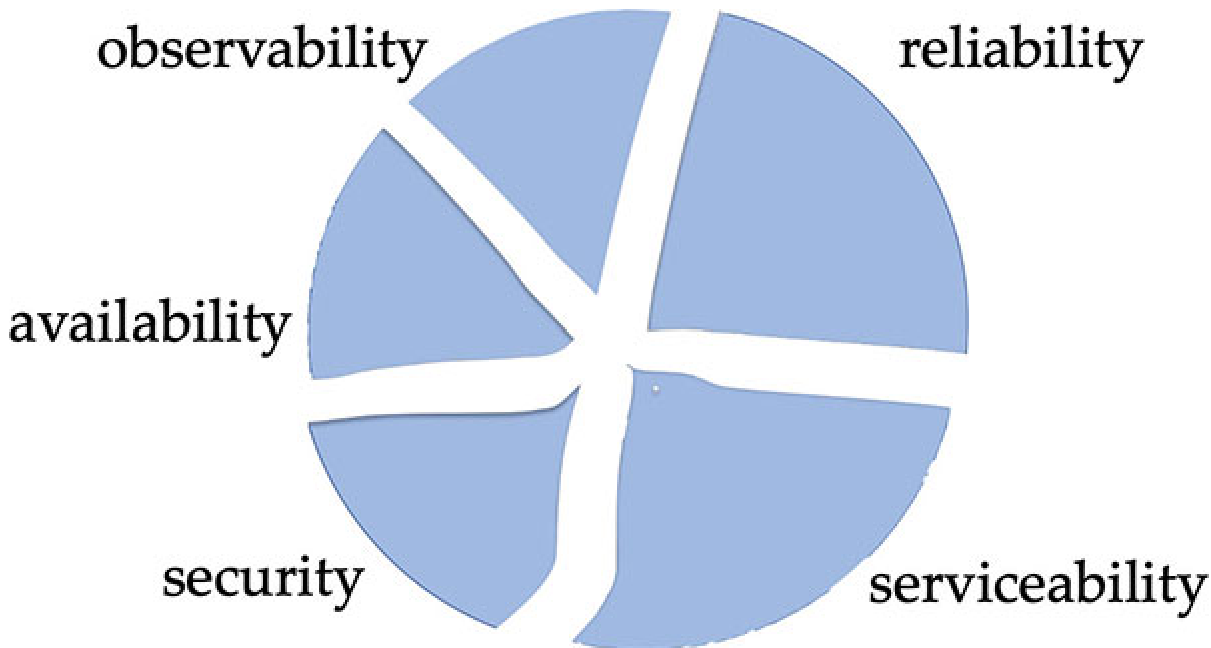

Figure 3.11: *A graphical view of common production-grade software capabilities*

Best practices

The common serviceability best practices are discussed here. You need to ensure that:

- The server software has a well-defined lifecycle in the design. This helps scope and understand anomalies based on problem reports.
- There is well-defined failure data for every possible anomaly type. The known anomaly types of a web server are crash (abnormal

termination), hang (low CPU), tight loop (high CPU), exceptions, performance degradation, memory leak, memory exhaustion, and incorrect result from computations.

- There is well-defined failure data capture method for every possible anomaly type.
- There are reliable tools for every possible anomaly type.
- There are proven methodologies for every possible anomaly type.
- The hosting system is prepared for failure data capture upon the occurrence of anomaly events before deployment.
- The design of the server software is inherently accommodative of serviceability. This means the high-level design review also has a checklist on serviceability confidence capture.

Web server considerations: Observability

Observability is defined as the ability to understand the state of the application through its naturally collected data points, without resorting to debugging means. The state of the application would mean the health of the process, the load on the system, the internal state of the language runtime/virtual machine, the client interaction patterns, the request-response demography, the fault rates, the resource usage patterns, and so on.

Importance of observability

Why is observability important? As a web server adopts other above-mentioned best practices (for example modularity, security, maintainability, etc.), the complexity of the software increases, and human examination and assertion becomes difficult and error prone. At the same time, understanding how efficiently we designed our software is important for its subsequent enhancement and evolution. For example, an aggregated time graph may show us that a specific microservice in our application is a bottleneck with respect to request traffic and would benefit from independent scaling.

Best practices

Here are some of the observability best practices:

- Understand the key observables (data points). What do we want to observe in order to assert the key performance indices?
- Define the interpretation of data points. What data will help us make meaningful assertions?
- Understand the life span of data points. How much of such data will help us make these assertions with reasonable confidence?
- Define lifecycle for data points. When does data collection start, when does it end, and where is the data stored? How long do we retain the data?
- Define a comprehensive data collection method. How do we collect data from various subsystems and components of the server software? Define a unique collection method for better isolation and control over these methods.
- Define a comprehensive data aggregation method. How is the collected data aggregated? What aggregations are meaningful to the user?
- Define a comprehensive data visualization method. How is the data visualized? Is the visualization easy to interpret? For example, is the given data better represented through a pie chart as opposed to a bar graph?
- Document observability artifacts in the software. This helps fix bugs and enhance, extend, and control the observability-related code in our software.

Conclusion

In this chapter, we learned the basic concept of a web server and defined the server's core components. To scope the server's context in the wide spectrum of software architectures, we made a comparison of our web server with a desktop application that serves a similar function. This comparison helped us observe the characteristic features of a request processing system when it is situated in the backend, across the network, at a central location. This, in turn helped us visualize a number of tradeoffs with respect to key software efficiency parameters, examine the fundamental considerations of a web server developer, and learn about best practices for an efficient web server, addressing the issues that emanated by

virtue of its placement and accessibility. In the next chapter, we will start putting the theory into practice. We will write a simple web server that responds with the time of the day and try to make meaning out of it.

CHAPTER 4

Our First Program: Time of the Day Server

So we have obtained a detailed background on Node.js programming by now – the fundamental principles of event-driven architecture to set the premise of Node.js as well as many considerations of web server applications. In this chapter, we will start putting the theory into practice. We will write a simple web server that responds with the Time of the Day and illustrates each program part.

Structure

In this chapter, we will cover the following topics:

- The hello world web server program
- A Time of the Day program
- Running the server
- Program parts

Objective

After studying this chapter, you should be able to understand the basic ingredients of a Node.js program. More specifically, a web server program. We will pick up a trivial 'hello world' program to demonstrate the development lifecycle and then move on to the Time of the Day server. We will then examine specific parts of the program and learn what those mean and how each piece is contributing to the overall server logic.

A 'hello world!' server

Just like any other program, we start with a 'hello world' program. This helps us clearly identify parts of the program. Incremental understanding

comes in handy, as some level of understanding is already available for a 'hello world' program for any programmer.

Here's a 'hello word' program written as a Node.js web server code:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   r.end('hello world!')
4. })
5. s.listen(12000)
```

Running the program

Follow the given steps to run the preceding program:

1. Save the program as '**`hw.js`**'.
2. Open a terminal and go to the location where the file is present.
3. Run the code as follows:

```
code — node hw.js — 35×6 — ⌥⌘2
#node hw.js
```

Figure 4.1: *Running the 'hello world' web server program*

Accessing the server through the browser

Now, while our program is running in the terminal, open a web browser and access this URL: **http://localhost:12000**, as shown in the following diagram:

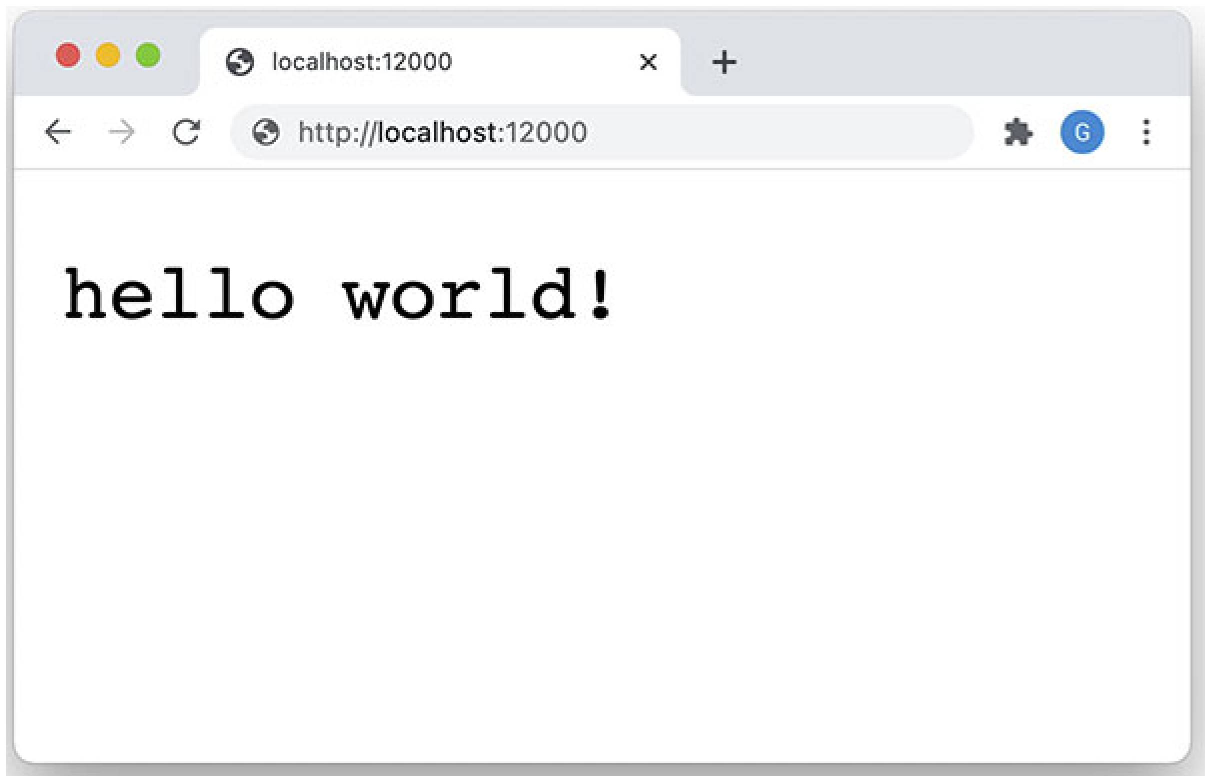

Figure 4.2: *Accessing the web server through browser*

That was our first Node.js program! We will come back to it later in this chapter to inspect different parts of it. For now, we will examine a Time of the Day server.

> **Assignment:** *Change the message that the server sends to some other message, restart the server program, refresh the request in the browser, and observe the response. Do you see the change that you expected?*

A Time of the Day Server

Now, let's build our Time of the Day server that responds to its client with the current time:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   r.end(new Date().toString())
4. })
5. s.listen(12000)
```

Just like earlier, run this program in the terminal, as shown:

```
code — node tod.js — 36×6 — ⌥⌘2
#node tod.js
```

Figure 4.3: *Running the Time of the Day web server program*

And then access it through the web browser, as illustrated:

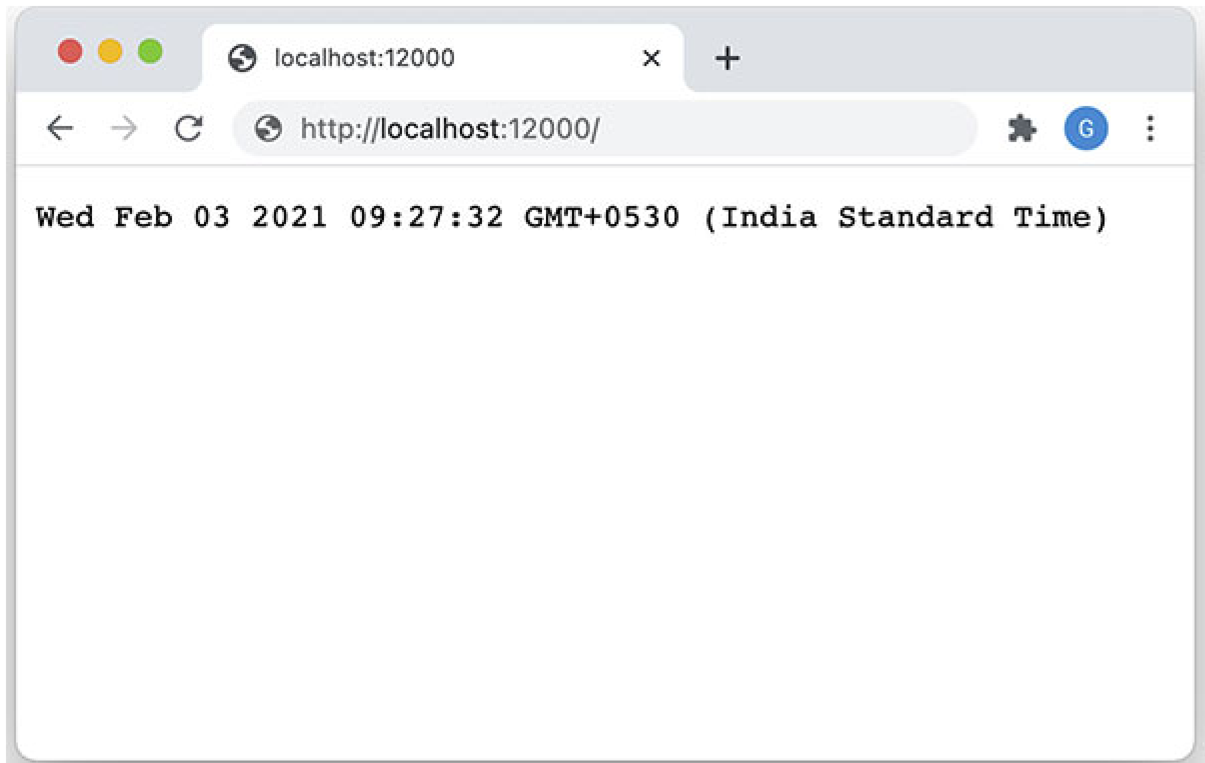

Figure 4.4: *Accessing the Time of the Day server through browser*

That was simple! We should appreciate that 'hello world' and Time of the Day were identical, except the message composed at the server side and passed to the client.

If you take a closer look when string is displayed in the browser, it brings up this question: Is the time with respect to the server or the client? It is the server's time, as the time was captured in the server's code, through this line:

```
1.   r.end(new Date().toString())
```

Of course, if we are accessing the client in the same system where the server is running, there is only one time. But think about a more real-world scenario where the server could be located elsewhere in the network.

The fact that it shows the server's local time imposes two issues:

- The user has to make adjustments with respect to the server's time zone
- It accidentally reveals where the server is located (an unwanted security exposure)

A more reasonable approach is to 'hide' the server's time zone and show an absolute time that every other locale can use as a reference, such as **Universal Co-ordinated Time** (**UTC**).

So our modified code uses a `Date` API that converts the local time to UTC time:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   r.end(new Date().toUTCString())
4. })
5. s.listen(12000)
```

Just like earlier, run this program in the terminal, as follows:

```
code — node todutc.js — 35×6 — ⌥⌘2
#node todutc.js
```

Figure 4.5: Running Time of the Day with UTC

And then access it through the web browser, as shown:

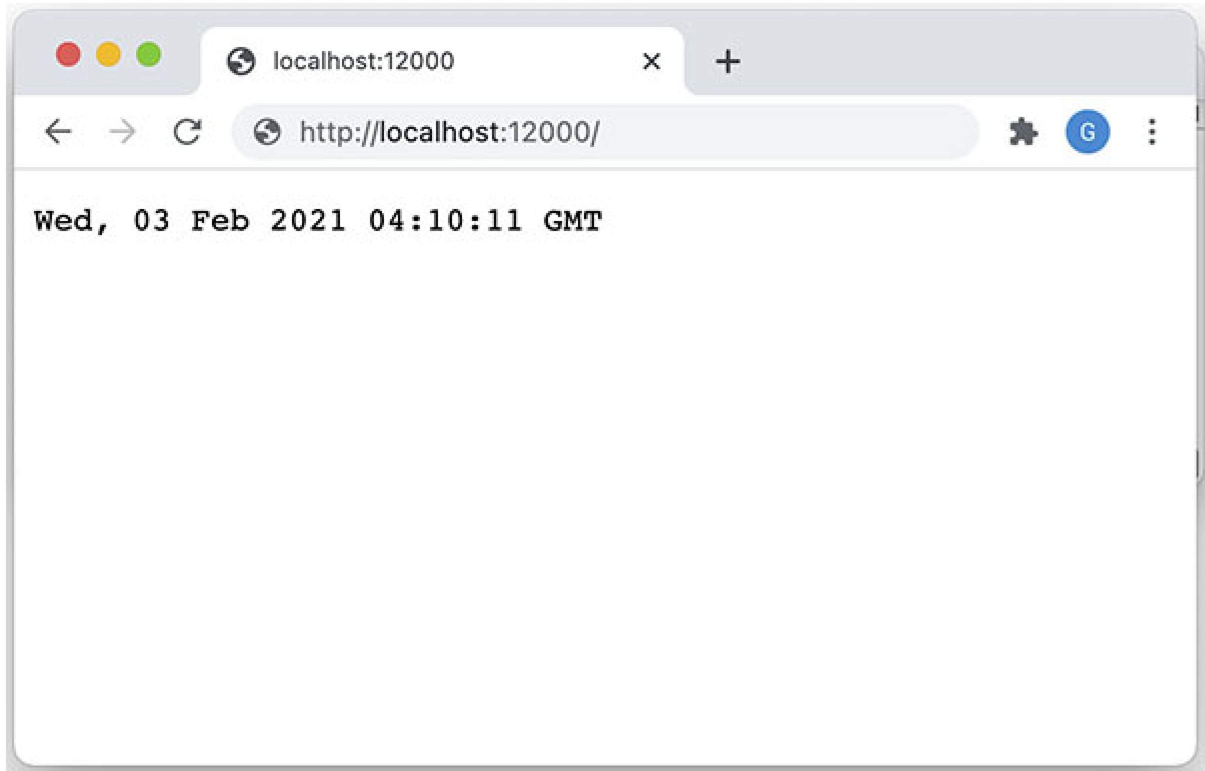

Figure 4.6: *Accessing the Time of the Day server (with UTC) through browser*

Now, as you can see, the time is displayed in UTC, solving both the mentioned issues. The user has no awareness about where the server is, and they can compute the local time using well-known arithmetic on UTC to obtain their locale-specific time.

Program sections

At a higher level, there are five discrete sections in a simple Node.js web server. Let's look at each of those.

Require statement

The '**`require`**' function is an essential part of Node.js API. It is a global function used to include, incorporate, or import reusable modules into your program. A module is a code unit that exports one or more fields and/or methods. The caller of 'require' gets an object that contains the exported content from the module.

Node.js APIs are made available as modules so that they can be easily used in a program.

In our code, the first line is:

```
1. const h = require('http')
```

It indicates importing a module called **`'http'`** into our program. After this statement, the **`'http'`** module capability is available through variable 'h', and our program has access to the **`'http'`** module, which comes in handy for creating an 'http' server.

While 'http' is the most common, it is not the only module that abstracts server capabilities. In our next chapter, we will make comparisons between the 'tcp' and 'http' protocols and understand the difference in server implementations with these two protocols.

Assignment: Print the content of the 'http' object to the console and try to identify some fields and their meanings.

The server loop

This is the most important part of the web server. The loop code is shown as follows:

```
1. const s = h.createServer((q, r) => {
2.   r.end(new Date().toString())
3. })
```

As we can see, it has two sections:

- The server creation
- The client handler

The **`'createServer'`** API of the **`'http'`** module is used to create an **`'http'`** web server. The create object is returned in the variable 's' that we can use for managing various lifecycle operations of the server.

The anonymous function passed as the single argument to **`createServer`** is the client handler callback (in some programming world, this is called a servlet). This handler is responsible for handling a client connection.

The contract that an 'http' server makes is that:

- This handler function is invoked every time a client connects
- The two input parameters 'q' and 'r' are populated upon entry
- 'q' is an abstraction around the client request (contains the request header, body etc.)
- 'r' is an abstraction around the server response (means to compose and send response)

This function is re-entrant in that when multiple clients connect at the same time, this function would be invoked for each one independently, with no side effect on the others.

This function roughly corresponds to the lifecycle of a client connection: the request–response cycle. Upon entry to this function, we can say that a new client request has arrived, and its request is completed when this function returns after sending a response.

The request and response objects that encapsulate the client and server respectively, represent classical web server programming and follow the same abstraction that is followed in many other web programming platforms.

Both the request and response objects are streams (Stream) as well as event emitters (EventEmitter) so that they can manage flowing data and define several events that make up the lifecycle of request and response.

In our next chapter, we will make a detailed examination of the request and response objects, including their inheritance, composition, and lifecycle

events.

Note: Heavy parsing of request happens behind the scenes before the request object is populated. This is because the client request originally comes as a binary stream of data over the network, formatted according to the selected protocol. This needs to be first parsed, then verified for conformity to the protocol specification, and then transformed to the structure aligned with that of the request object.

The date

Another component in the program is the date computation. ‘Date’ is a standard JavaScript class, so **`'new Date()'`** provides the current date with time, with respect to the invoking system’s locale. There are a number of functions that help creating as well as representing a date object in various convenient ways.

The server response

The server’s response is sent to the client through the ‘**`end`**’ function, as shown:

```
1.  r.end(new Date().toUTCString())
```

If there are large chunks of data to be sent to the client, the ‘**`r.write()`**’ API can be called as many times. But calling ‘**`r.end`**’ ends the response, and no more response can be sent through to that client through that connection.

Most browsers start rendering the page only when it receives the final piece of data. Due to this, the `**end**` API is essential.

Questions: Comment out the response sending code in our program and access the page through the browser. Observe what happens to the browser. Why?

The server listen

The listen API is invoked as follows:

```
1. s.listen(12000)
```

This is the last piece of code in our program. The '`listen`' API of the server object causes the server to prepare itself for receiving client connections. In the most common form, it takes a port number where the server should be available. The same port number should be used by the clients when they compose the server's URL.

Note: A port number can be listened to by only one process at a time.

Conclusion

In summary, we studied each line of our program and understood their meaning and purpose in the context of a web server that provides current time information to a requesting client. In that process, we also understood some of the powerful features and APIs of Node.js that make it a flexible and easy web programming platform. In the next chapter, we will take a closer look at the networking features of Node.js around web applications. This will include API abstractions for different protocols, data transport mechanisms, and client-server interactions.

Exercises

Two common yet simple problems are described below to get you up to speed with the practical aspects of web server programming.

Problem #1

In a particular scenario, the following error occurred when the Time of the Day program (either of the versions) was run on the terminal:

```
#node tod.js
events.js:291
      throw er; // Unhandled 'error' event
      ^

Error: listen EADDRINUSE: address already in use :::12000
    at Server.setupListenHandle [as _listen2] (net.js:1316:16)
    at listenInCluster (net.js:1364:12)
    at Server.listen (net.js:1450:7)
    at Object.<anonymous> (/Users/gireeshpunathil/Desktop/book1/4/code/tod.js:5:3)
    at Module._compile (internal/modules/cjs/loader.js:1251:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1272:10)
    at Module.load (internal/modules/cjs/loader.js:1100:32)
    at Function.Module._load (internal/modules/cjs/loader.js:962:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
    at internal/main/run_main_module.js:17:47
Emitted 'error' event on Server instance at:
    at emitErrorNT (net.js:1343:8)
    at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  code: 'EADDRINUSE',
  errno: -48,
  syscall: 'listen',
  address: '::',
  port: 12000
}
#
```

Figure 4.7: *Demonstration of potential server startup error*

- What does the error mean?
- Why does this error occur?
- How to resolve the error?

Problem #2

In a particular scenario, the following error was observed when the Time of the Day program (either of the versions) was run and accessed through the browser:

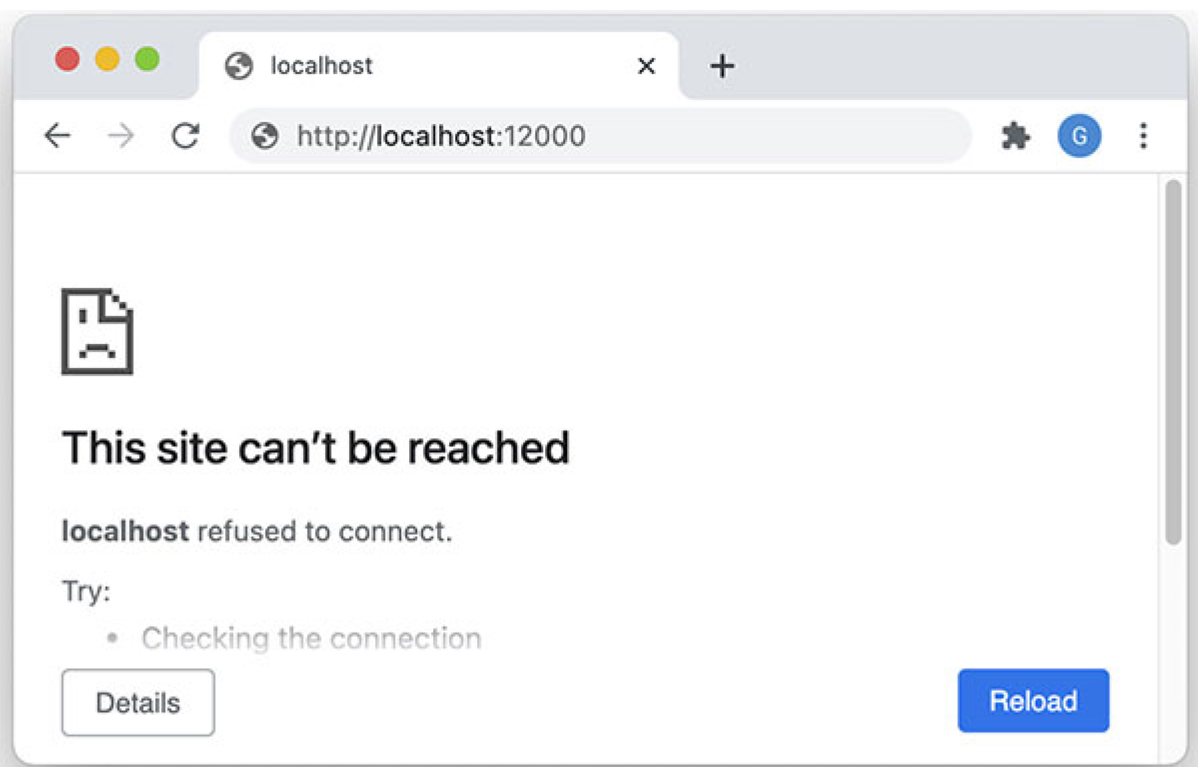

Figure 4.8: *Demonstration of potential access error at the client*

- What does the error mean?
- Why does this error occur?
- How to resolve the error?

CHAPTER 5

Common Networking Interfaces of Node.js

At this stage, we are ready to dig deeper into the API abstractions that Node.js offers to implement common web applications. We will first compare the raw native networking protocol TCP/IP with a higher-level, convenient protocol HTTP. This is important to justify our choice of HTTP for building applications throughout this book. We also need to understand how data transport works in the underlying network. Then, we will look at the streaming APIs, as almost all client-server communication is stream-oriented, and buffer APIs, as almost all stream-based data transport uses buffers to hold the data. We illustrate the two popular web server abstractions—'*request*' and '*response*'—that represent the fundamental building block objects for the client-server interaction. We will also examine in detail the configurations that affect the server's behavior and the server's life cycle control points and server events. Additionally, we will touch base upon a few other important networking APIs that we will use in this book.

Structure

In this chapter, we will cover the following topics:

- Network programming
- TCP versus HTTP
- Node.js streams
- Node.js buffers
- Request and response objects
- Request and response life cycle
- Server configuration
- Server's life cycle events
- Other networking APIs

Objective

After studying this chapter, you will be able to understand the common network abstractions in the Node.js programming platform. These include network programming considerations, the necessity of network protocols, network data transport considerations, and how Node.js abstracts these capabilities, aligning the APIs well with the underlying philosophy of event-driven and asynchronous programming style while comprehensively covering all the aspects of the said capabilities. You will be able to appreciate the current server abstractions as a comprehensive set of APIs and understand why streams and buffers are extensively used in network programs. You will also learn about server configurations and server life cycle events.

Network programming

Network programming is a programming paradigm in which computation occurs in more than one computing systems connected in a network. There are several programming models (topologies) that we can derive, depending on how the computations are organized, divided, aggregated, and consumed. For example, client–server, peer–to–peer, and so on.

The most common programming model of Node.js is that of a web server. It is a classic example of client–server topology and so, an example of network programming as well. We discussed the merit of this programming model in detail in *Chapter 3, Introduction to Web Server*.

A key difference between a simple desktop program and a network program, as explained in *Chapter 3, Introduction to Web Server*, is that the invocation and consumption of a computation function happens in-process (a direct function call) in the former, whereas it happens over the network in the latter. This difference implies mainly to the structure, functionality, and interface of the program. Among other things, we need a protocol to communicate between the processes across the network.

Why a protocol?

Earlier, we just mentioned the protocol; now is the time to look at it in detail. Let's start with the simple function call example. These things happen when the '`foo`' function calls the '`bar`' function, typically in a natively compiled program:

At the calling function side

- The function arguments are pushed to the program stack/designated registers
- The return address (where the callee should return) is pushed to the program stack
- A jump is made to the target function

At the called function side

- Creates a stack frame for holding its data
- Updates the stack pointer
- Executes its code
- Resets the stack pointer
- Jumps back to the return address
- The execution continues in the caller

Here's a concrete example with a C function 'foo' making a call to 'bar'. Let's look at its source code and the generated assembly in a 'x64' system and understand how the call is being managed.

The following is the code for the caller function '`foo`' and callee function '`bar`':

```
1. void foo() {
2.   int ret = bar(0xaaaa, 0xbbbb, 0xcccc);
3. }
```

```
1. int bar(int p, int q, int r) {
2. return p + q + r;
3. }
```

Now, let's also look at how the generated machine code will look for these functions. Only the relevant portions are displayed for simplicity and clarity. Also, I have annotated every instruction with comments, for better understanding of what is happening at the lower level.

The following code shows the call site of 'bar' in 'foo':

```
1. // store the third argument into edx
```

```
2. mov     edx,0xcccc
3. // store the second argument into esi
4. mov     esi,0xbbbb
5. // store the first argument into edi
6. mov     edi,0xaaaa
7. // invoke the target method
8. call    bar
```

Here, the caller's argument (or parameters when they are referred in the callee) are stored in designated registers, and then the target method is invoked. In some other architectures, these arguments may be pushed onto the stack directly or stored in other registers or a combination of both.

In the 'x64' architecture, the 'call' instruction is internally composed of two actions: i) push the address of the next instruction to the stack, ii) jump to the mentioned target.

Now, let's look at what happens in 'bar', the callee. Here's how the first part of the function will look:

```
1. // store the current frame base pointer
2. push    rbp
3. // set the current stack pointer as the new frame base
   pointer
4. mov     rbp,rsp
5. // store first param into first slot of base pointer
6. mov     DWORD PTR [rbp-0x4],edi
7. // store second param into second slot of base pointer
8. mov     DWORD PTR [rbp-0x8],esi
9. // store third param into third slot of base pointer
10. mov     DWORD PTR [rbp-0xc],edx
```

In the callee, the base pointer register is saved on the stack first, and then the current stack pointer value is stored in the base pointer, marking a new stack frame. Then, each of the parameters (which were stored by the caller) are stored at designated slots in the memory, with 'rbp' as the reference, at specific

offsets from it. This whole exercise, performed before the business logic of the function starts, is called function prologue.

The following code shows the actual 'body' of the function – the adding logic:

```
1. // load first param into edx
2. mov    edx,DWORD PTR [rbp-0x4]
3. // load second param into eax
4. mov    eax,DWORD PTR [rbp-0x8]
5. // add eax and edx, store the result in edx
6. add    edx,eax
7. // load third param into eax
8. mov    eax,DWORD PTR [rbp-0xc]
9. // add eax and edx, store the result in eax
10. add    eax,edx
```

This is self-explanatory. The values are fetched from the registers, and the add operation is performed one by one. At the end, the result is stored in the 'eax' register, also known as the **accumulator register**, which holds the return value from a function.

The following code shows the final piece of the callee, called epilogue:

```
1. // unwind the frame, get the old base pointer back
2. pop    rbp
3. // return, leaving the result in eax
4. ret
```

The action performed in the epilogue is to restore the base frame pointer and thereby, unwind the current frame and restore the previous frame before returning to the caller. When the execution reaches the caller, everything is back to normal at the time of the call, with the return value of the call available in the 'eax' register. The caller continues from there.

This set of rules is called calling convention or subroutine linkage convention. It is part of a wider rule specification called **Application Binary Interface** (**ABI**). If any of these steps are slipped or violated while a program is under execution, the program will malfunction, mostly leading to a crash.

Note: A computer CPU works based on a specified behavior, and this specification is called its architecture. Among other things, the two important aspects of a machine architecture that are relevant here are: i) the calling convention or subroutine linkage convention, and ii) instruction set specification.

TCP/IP

Clearly, we need a rule when a program is talking to another program. The complexity of the rules increases when programs are situated across the network. For example, when program A passes a piece of data to program B across the network, there are several questions, like:

- How can we uniquely identify program A's data? (hint: network has access to all the posted data)
- How do we know program A's data boundary/length? (hint: data flows as octet streams)
- How do we know program A's data sequence/order? (hint: which one is the higher order bit?)

If every application starts dealing with these complexities, network programming will become extremely complex and unmanageable. Additionally, programs lose inter-operability. So, it is natural that we define a set of rules that comprehensively cover how communication happens between programs, while the programs can better focus on their 'business logic'. TCP/IP is the most commonly used set of such rules. While these are two discrete protocols (Transmission Control Protocol and Internet Protocol) layered on top of each other, we don't need to understand the details and differences. Also, there are many other protocols over and below and also at par with TCP/IP that are outside the scope of this book.

Note: Here's a simple example that gives us a feel about the importance of a network protocol. At the lowest level, when two-byte data (B1, B2) reaches an endpoint, the question as to how to combine the bytes to reconstruct the original word—B1B2 or B2B1—is defined by a concept called endian-ness. Two types of endian-ness are prevalent: big endian and little endian.

A network protocol defines a set of rules for transporting data in the network. TCP/IP provides reliable, stream-oriented data to participating applications in

the network. By communicating over TCP/IP, the programs are spared from dealing with data transport complexity like data chunking, routing, traffic load balancing, error correction, managing lost packets, and aggregation. Stream-oriented means a hand-shake and connection needs to be established between the endpoints before transferring data.

In other words, the contract with TCP/IP is that a piece of data 'hello' sent from a network—endpoint A—reaches its destination—endpoint B—intact, as 'hello' and A and B are connected through TCP/IP way.

The following figure shows this contract in a symbolic manner:

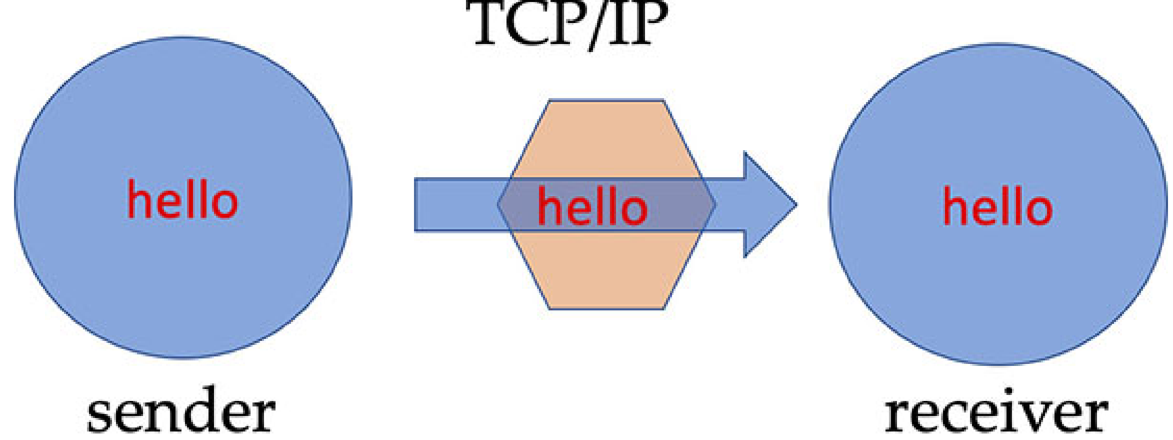

Figure 5.1: *A graphical view of TCP/IP's contract*

An organization called **International Standards Organization** (**ISO**) defined the reference architecture for data exchange between connected computers. This architecture is called **Open Systems Interconnection** (**OSI**). It takes the shape of a connected set of pipelines in the computing abstractions, starting from the application to the physical network apparatus for the forward data flow (writing), and starting from the physical network apparatus to the application, for the reverse data flow (reading).

The following figure shows the seven layers of OSI data interchange architecture:

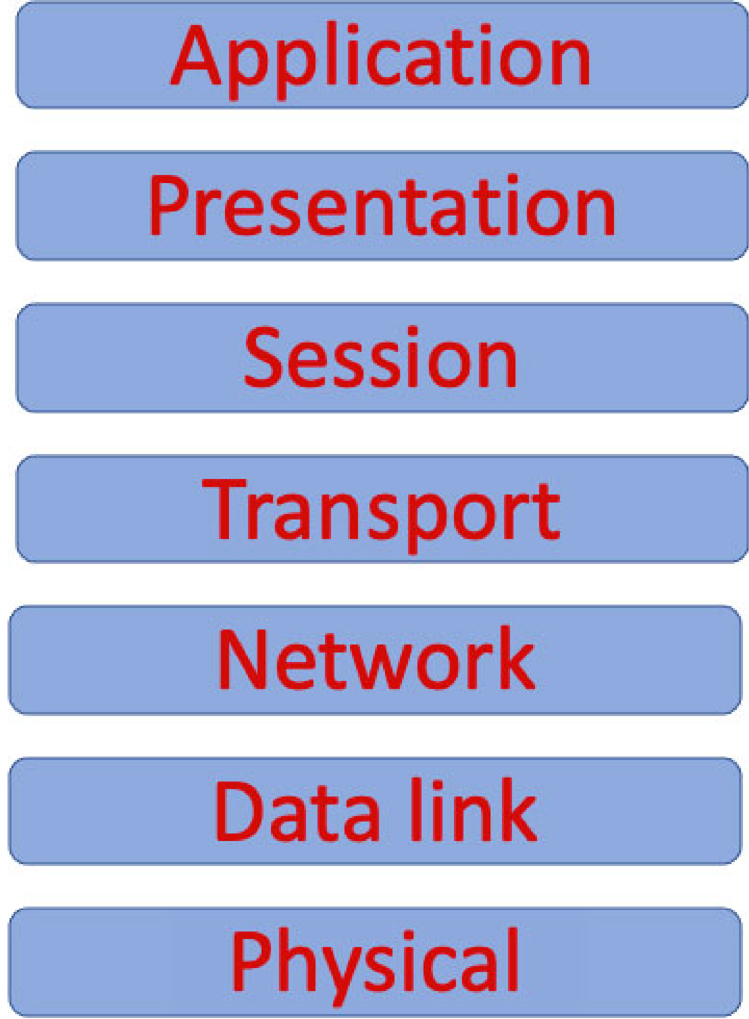

Figure 5.2: *The OSI-specified networking layers*

While the OSI model's seven layers specify a guideline, TCP/IP is a concrete implementation of the same, and at the implementation level, a few layers have been coalesced by virtue of convenience and modularity. In other words, TCP/IP is a protocol of the actual Internet that we use and is heavily derived from the OSI specification. Here's a brief description of the TCP/IP layers:

- **Application**: Defines the standard internet services and network APIs. A program deals with the network at this level. Examples: ftp and telnet. The standard network abstractions such as sockets and ports are part of the application layer.
- **Transport**: Defines data integrity and reliability constraints in forward and reverse paths.

- **Internet**: Defines transport protocols of data packets in the network: routing, resending lost packets, and so on.
- **Physical**: Defines the characteristics of the hardware that transmits the data.

The TCP/IP programming interfaces, as implemented in the operating system and consumed by a standard C program, are listed as follows:

API	Meaning
socket	Creates an end point (server or client)
connect	Makes a connection with a remote (client)
bind	Binds with a port and network interface (server)
listen	Enables the socket to be usable (server)
accept	Waits for client connections (server)
read, write	Standard I/O APIs (server and client)

Table 5.1: *Core TCP/IP C APIs*

A socket abstracts the remote endpoint in a network program. In the server connected with a client, the socket that was created as a result of the connection represents the client. Similarly, in a client, the socket created as a result of the connection represents the server. In other words, if a client reads from a connected socket, it is actually reading from the server. This way, a reasonable two-way communication can be carried out using connected sockets.

Assignment: Write a simple client-server program in which the server echoes a piece of data that the client sends, in C language, using the above-mentioned APIs. Observe the complexity.

Fortunately, we don't need to dig deeper into the complexities of these concepts. In Node.js, we have sophisticated APIs that hide these complexities, and help us to focus more on our business domain, rather than in the network domain. Here's a simple example of TCP-based Node.js server and client that exchange a piece of word.

The following code illustrates a simple TCP server:

```
1. const n = require('net')
```

2. const s = n.createServer((s) => {
3.   s.end('hello world!')
4. })
5. s.listen(12000)

And the following code illustrates a simple TCP client:

1. const n = require('net')
2. const c = n.createConnection({port: 12000})
3. c.on('data', (d) => {
4.   console.log(d.toString())
5.   c.end()
6. })

And the following figure shows the execution of the code with the output:

```
code — -bash ▸ node — 35×6 — ⌥⌘2
#node tcp_server.js &
[1] 38938
#node tcp_client.js
hello world!
#
```

Figure 5.3: *TCP client server execution output*

As we can easily note here, the complexity of the data transported in the network is not visible in the program! Instead, a sophisticated abstraction is made available to the program called '*socket*', which is used to perform the communication. The variables that hold the return values of both '**`createServer()`**' in the server and '**`createConnection()`**' in the client are sockets, which are used for communication.

HTTP

Is that enough for writing large-scale network programs? Does it (TCP/IP) satisfy our need to communicate easily and seamlessly?

Yes, to an extent. But the protocol is still low-level to be easily consumable for software that operates at much higher levels of abstraction. A simple example is a case wherein an application form is submitted on a website.

The following figure shows a sample form data:

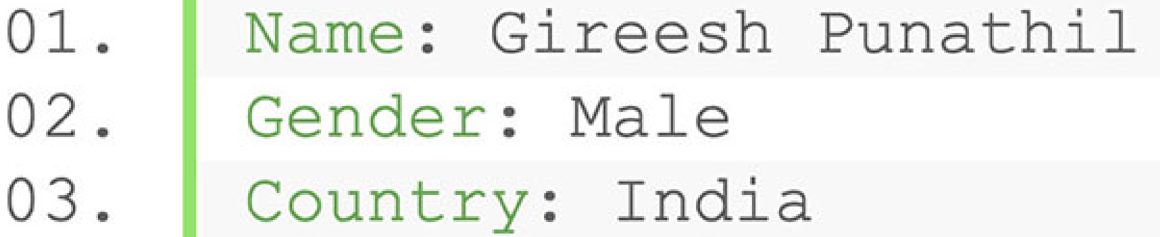

Figure 5.4: *A sample form data*

Now when the form is filled and submitted in the client browser, it needs to be transmitted across the network and parsed into higher-level programming constructs on the server side. How will we easily manage that transportation and transformation?

- We need a rule that defines the data is being sent to the server from the client
- We need a rule that defines what type of data is being sent
- We need a rule that defines the actual record length
- We need a rule that defines how individual entries are delimited
- We need a rule that defines how key-value pairs are delimited

Again, if each application starts building these set of rules privately to the application in a custom manner, the development becomes complex. Plus, applications lose their inter-operability.

Question: Take a step back and reflect on where these complexities came from. Did a function invoking another function in the same program have these issues to tackle? Or is it that they also have complexities but at a different operating space?

So it makes perfect sense to define another set of rules that solve the same problem as that of TCP/IP (standardizing data transport mechanism) but

operate at a much higher level, which is closer to the application's level of abstraction.

Hyper Text Transfer Protocol (**HTTP**) is an application-layer protocol that operates on top of the TCP/IP protocol, defines a set of rules for transporting data in a more flexible and seamless manner than the more complex and low-level TCP/IP. The protocol includes rule definitions for request type, message format, message body, transfer encoding, and status code. With these primitives, it becomes easy for an HTTP parser to interpret the data. Further, it becomes easy for an HTTP endpoint to receive and handle the request and send the response.

When the above-mentioned record is transferred through HTTP protocol and intercepted in the network, it will look as follows:

```
01. POST / HTTP/1.1
02. Host: myhost
03. Content-Type: application/x-www-form-urlencoded
04. Content-Length: 47
05.
06. Name=Gireesh Punathil&Gender=Male&Country=India
```

Figure 5.5: *A sample HTTP payload*

The contract with HTTP is that a piece of data 'hello' sent from a network—endpoint A—reaches its destination—endpoint B—intact as 'hello', with additional information such as the request type, the request context, the data type, and the data length, if A and B are connected through the HTTP protocol.

The following figure shows this contract of HTTP in a symbolic manner:

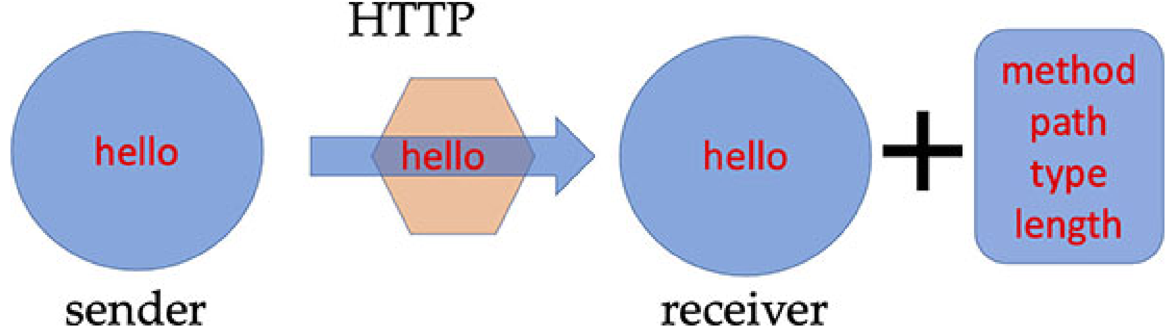

Figure 5.6: *A visualization of contract with the HTTP protocol*

The HTTP protocol deals with the important things:

- What to do with the data (the request method)

- Where to use the data (the request path)
- Who the data is meant for (the host)
- Its type (the Content-Type)
- Its length (the Content-Length)

Now that the data has its raw message content as well as a number of attributes around it, the recipient can understand the ‘intent’, through standard mechanisms, without resorting to low-level primitives. In the below sections, we look at each tokens in the HTTP message, and interpret the mentioned example HTTP message format.

POST

This token specifies the HTTP method or request type. The request type defines the nature and format of the rest of the request body. POST indicates that the client is providing a block of data, along with the request – creating or updating a resource. Other common HTTP methods include `GET`, `PUT`, and `DELETE`.

/ verb

This token specifies the request target or protocol path. The path defines a context of the request such that different requests can be characterized at the server, classifying them based on the request target.

Different forms of the path include:

- The simplest form, such as ‘/’
- A simple token path, such as ‘`/foo`’
- A resource path form, such as ‘`/cat.png`’
- A query string form, such as ‘`/test?name=foo&type=bar`’

HTTP/1.1

This token specifies the HTTP version in use. Naturally, with a different version, one would expect a change in the format for the rest of the message. Also, specifying the version is a request to the server to format its response accordingly so that the client can consume it.

Host

This token specifies the host name and port number of the server to which this request is sent. If the port number is missing (as is the case here), it would mean the target port is the default port (80).

Question: What purpose the 'Host' field solves? If the message is destined to a specific host, and this field helps with that, how did the message reach this server in the first place, before parsing the field? Also, if the need is to select the destination, who should be the consumer of such a data? the lower level TCP stack, or the application itself?

Content-Type

This token specifies the data type of the content part of the request. It has two parts: the media type that specifies the high-level type, and the media subtype that provides a hint about the file extension. There is a long list of valid data types: application, audio, video, image, multipart, text, and so on.

Content-Length

This token specifies the number of bytes of the request body, including the metacharacters that are used to delimit records.

And finally, the body itself is composed as a single string by combining all the message parts with well-defined delimiters separating them.

There are many more fields, each with several extensions and exceptions, but we are good here in terms of understanding the basic format of a typical HTTP message.

Do we see an evident flexibility improvement here in terms of dealing with large and complex business data? Yes, we can. As a result of these improvements, HTTP is the preferred way of communicating for common web applications.

Tip: How do we see an HTTP message header? A simple technique is to write and run a TCP server that just prints the incoming request and access the server through a browser client as if it is an HTTP-based server. The client will fail, but the message in the TCP server will be the complete HTTP message that the browser has sent.

Now let's revisit the questions we could not solve with TCP alone and see how they fare in the context of HTTP capabilities.

- We need a rule that says the data is being sent to the server from the client: this is now addressed, as we have a 'method' verb defined in the HTTP header, such as 'GET' or 'POST'.
- We need a rule that defines what type of data is being sent: this is now addressed, as we have a 'Content-Type' verb defined in the HTTP header – such as 'application/text' or 'image/jpeg'.
- We need a rule that defines the actual record length – this is now addressed, as we have a 'Content-Length' verb defined in the HTTP header.
- We need a rule that defines how individual entries are delimited – this is now achieved, as we have a well-defined delimiter for the individual fields in the message, such as '&'.
- We need a rule that defines how key-value pairs are delimited: this is now achieved, as we have a well-defined delimiter for the key-value pairs, such as '='.

Evidently, dealing with HTTP is much simpler than TCP/IP. The latter gives fine control on the data being transported but is more complex.

Question: In Node.js, when the server receives an HTTP message (in the form of a request), it is not in the form explained above. Instead, the individual values are decomposed and constructed into 'request' object. Who performs this activity and when? What is this action called?

In summary, we learned the definition of network programming and the associated complexity. We then understood the need for protocols and studied the two most popular network protocols: TCP/IP and HTTP. We also made head-to-head comparison of the two in the purview of a web program to gauge its efficiency with both.

Node.js streams

Now that we have a well-defined protocol in place for communicating, let's take peek at the data to be communicated. Earlier in this chapter as well as in *Chapter 3, Introduction to Web Server*, we touched upon an important concept. Let's recollect that in detail.

The flow of data has implications on the transport layer as well as the application layer. Let's see how:

First, data is chunked, and then the chunks are split into packets. The packets are then sent across the network, not necessarily in a specific order or through a specific route. These packets are regrouped at the destination and ordered. The ordered data is coalesced (de-chunked) back as required and then presented to the application. The volume and speed of the data will directly influence these sequences. This is the implication of data flow on the transport layer.

Note: Data is chunked because it has to pass through socket buffers (internally stored in kernel memory), which have a definite size. Chunks are split into packets, as TCP/IP packets also have a definite size, which is much lesser than the size of the socket buffer.

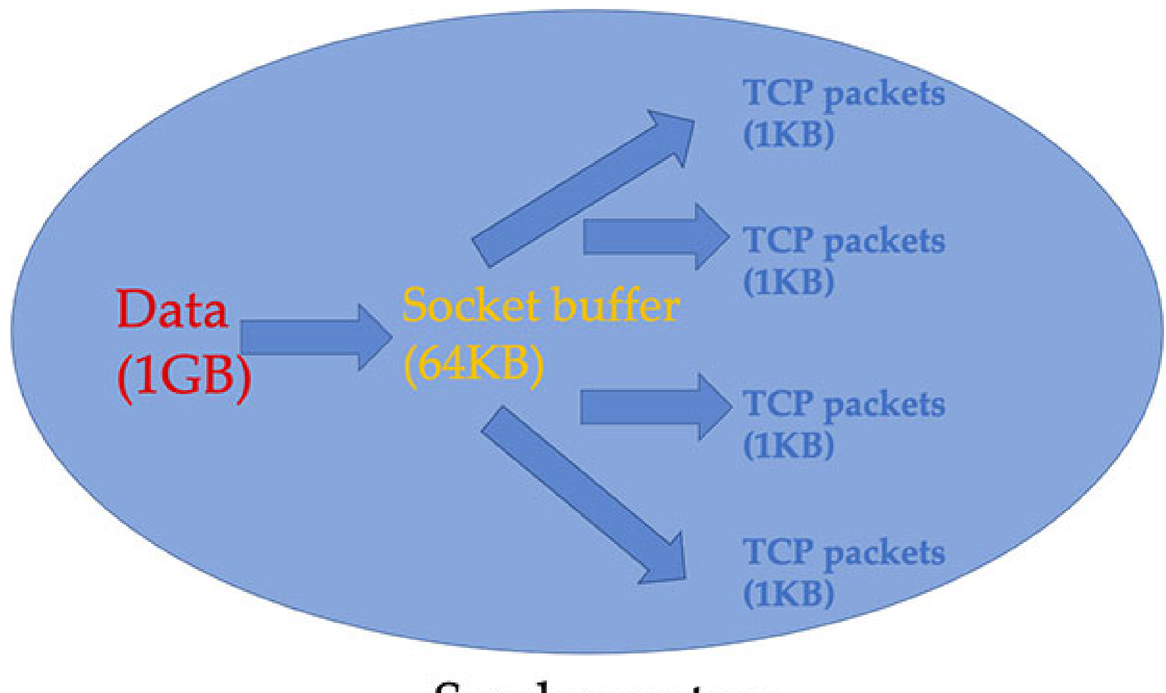

Figure 5.7: *View of network data fragmentation*

Second, consider two network endpoints engaged in a data transfer. When an endpoint has initiated a read from the socket, these are the possibilities for the state of the other endpoint:

- The other endpoint has already written its data
- The other endpoint is writing data

- The other endpoint is writing much more data than the receiver can handle
- The other endpoint is about to write data
- The other endpoint is not writing any data

The receiving end has to prepare itself to face these possibilities, or its function to receive the desired data will be adversely affected. For example, how can the data be managed if the other endpoint is writing much more data than the receiver can hold in a chunk? Similarly, what if the other endpoint is never going to write? Will the receiver have to wait forever? If so, what will happen to the program logic that depends on it? In summary, the volume, speed, and order of the data will directly influence these sequences. This is the implication of data flow to the application layer.

As we know, Node.js is single threaded, and we cannot afford the only thread dedicating itself to preparing for (or blocking) data transport in that platform. As we saw in the previous examples, the network data is already transported in a fluid manner, with no definite sequencing enforcement possible in any endpoint. These are two problems, but combining them gives us an innovative revelation—data transport over the network is a perfect fit for event-driven architecture!

That is, let the data flow as per the network transport considerations and constraints, but handle it incrementally when a certain amount has been received.

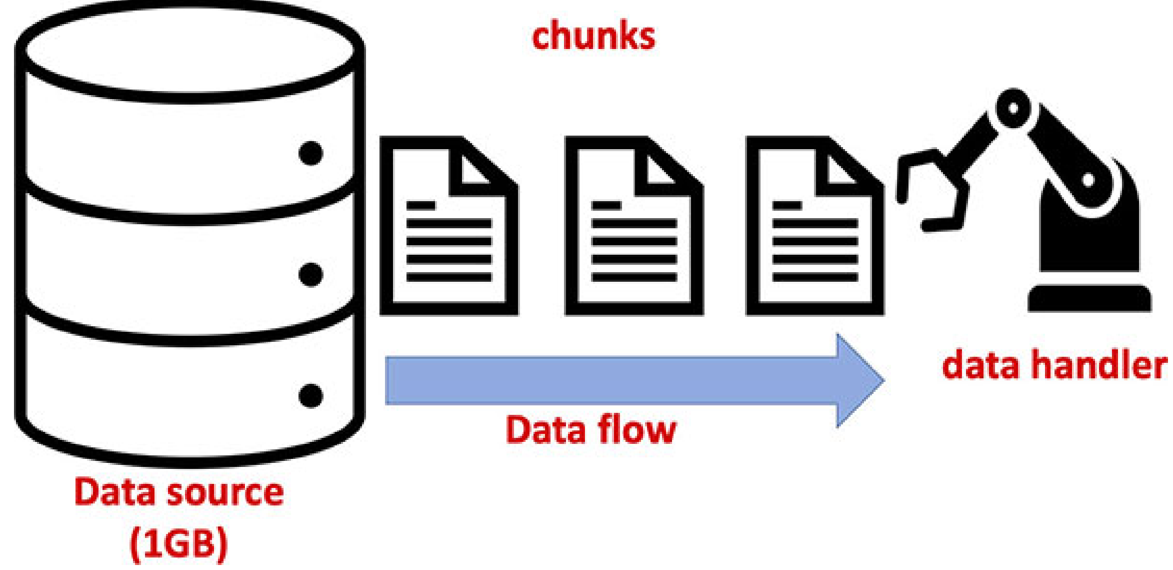

Figure 5.8: *Dataflow with streams*

Node.js stream is an abstraction around flowing data. Stream is not a Node.js-specific concept, but streamed data management is a natural alignment for event-driven architecture. With streams, functions are designed to operate with partial data. To mitigate the issue of additional processing because of operating on partial data, these functions are capable of producing partial data to a second stream. Such functions are called **filter functions**. Depending on how many levels of operations are required on the data, these streams can be connected sequentially, making a pipeline.

The following figure shows an example of a **Natural Language Processing** (**NLP**) pipeline that uses filter functions and pipes on stream data:

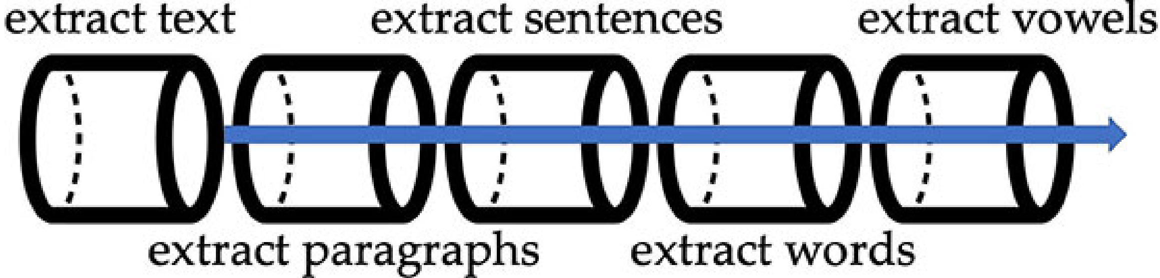

Figure 5.9: *A sample NLP pipeline that uses stream of data*

Data containers of all the I/O operations that Node.js implements are streams. This includes network, file, and console. In other words, you perform I/O operations on these devices through stream APIs. This gives an advantage to the programmer – uniformity across different data containers with respect to how we deal with their data.

Node.js stream has a relatively large set of life cycle events. Due to this, Stream is inherited from the 'EventEmitter' interface. This makes streams first-class event-driven objects. The next figure shows various Node.js objects that are event emitter types:

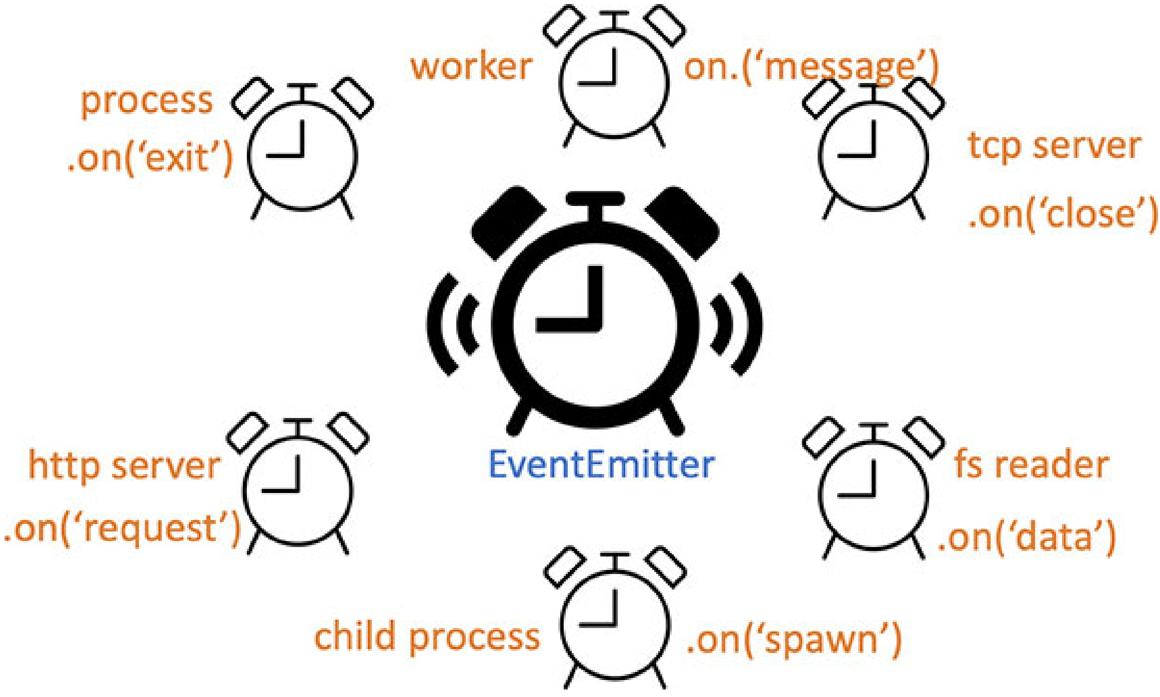

Figure 5.10: *EventEmitter and its common subclasses*

An added positive side effect of stream is memory efficiency. If you are dealing with a large amount of data, you can make do with less space by processing small chunks of data incrementally. The following code illustrates a small program that shows a simple usage of streams:

```
1. const f = require('fs')
2. console.log('the file content is:\n-----')
3. const r = f.createReadStream(__filename)
4. r.on('data', (d) => {
5.   console.log(d.toString())
6. })
7. r.on('end', () => {
8.   console.log('-----\nend of file')
9. })
```

In the program, we are opening our own source code as a stream and reading from it. ('__filename' is a predefined variable that can be used globally in Node.js; it represents the executing module's filename). The following figure shows what the output looks like when the program is executed:

```
code — -bash — 57×16 — ⌥⌘2
#node stream.js
the file content is:
-----
const f = require('fs')
console.log('the file content is:\n-----')
const r = f.createReadStream(__filename)
r.on('data', (d) => {
  console.log(d.toString())
})
r.on('end', () => {
  console.log('-----\nend of file')
})

-----
end of file
#
```

Figure 5.11: *Output of the sample stream program*

As we can see, the entire file contents are printed in the output.

Question: In the above-mentioned code, why is the data received in the data handler function 'stringified' before being printed? What would happen if the data is printed as is?

Node.js buffers

Picking up where we left off, streams are a convenient way to manage I/O data. One of the important life cycle events of stream object is 'data', which is emitted when there is a 'chunk' of data available in the stream for the program's consumption. The word 'chunk' deserves special attention here. If 1MB of data is sent from an endpoint A to another endpoint B, what is the size of 'chunk' when 'data' event occurs at B? Less than 1MB, exactly 1MB, or some system-default, such as 1KB, 64 KB, or 512 KB?

The answer is: It depends on a number of factors:

- The TCP implementation
- The TCP configuration
- The HTTP parser implementation
- The HTTP parser configuration

- The sender's kernel buffer size
- The recipient's kernel buffer size
- The speed of the involved network

Above all this, the stream API can take specific decisions as to when and with how much data to emit a 'data' event, such as:

- Wait for all the data to flow in over the network and then emit 'data' event
- Wait for a predefined amount of data (chunk) to flow in, and emit 'data' event for that
- Emit the event with whatever data came in in a single read from the network

There could be more possibilities with variants of the above-mentioned. The bottom line is that there are several considerations around what a meaningful amount would be in a single chunk, for which the handler can be invoked.

From the consuming program's standpoint, the length of a chunk in the 'data' callback is arbitrary for most practical purposes.

But wait! how can the data be chunked? If that happens, what does it imply for the underlying record? Remember the example message we used in the previous section? What if that message was broken into two chunks with the chunk boundary at an arbitrary number of characters from start, neither ending at a record boundary nor at a key or value boundary? The following figures show an HTTP message split at an arbitrary point due to the aforementioned considerations:

```
01.  GET / HTTP/1.1
02.  Host: localhost:12000
03.  User-Agent: Mozilla/5.0 Gecko/20100101 Firefox/78.0
04.  Accept: text/html,application/xhtml+xml,applica
```

Figure 5.12: *A chunked HTTP message part 1*

```
01. tion/xml;q=0.9,image/webp,*/*;q=0.8
02. Accept-Language: en-US,en;q=0.5
03. Accept-Encoding: gzip, deflate
04. DNT: 1
05. Connection: keep-alive
06. Upgrade-Insecure-Requests: 1
```

***Figure 5.13:** A chunked HTTP message part 2*

With this example, the receiving program will get the handler invoked twice – once with the first part of the message, and once again with the second. Evidently, the data cannot be reliably used by the program. This behavior is well understood and justified in an event-driven world. As a solution, we need to store the data flowing in until we can get a coherent unit of data so that we can start working on it.

What is the type of such a split data? With streaming in place, the incoming data usually has no type. Even if it had a definite shape and type at the source, the incoming data token cannot be cast to a system-defined or user-defined type as the data flows in the network as octet streams (byte streams) and is truncated at arbitrary points in a chunk.

If the original data was a record of three key-value pairs with 47 bytes length (forget the headers here), the incoming chunk at the receiving side in one iteration could be full 47 bytes, 25 bytes, or even 1 byte! The only known aspect of the data chunk is that it is octets.

How and where can we hold such data?

Node.js Buffer is a user-defined type to contain and process binary data of arbitrary length. At the lowest level, a Buffer is an array of unsigned bytes or unsigned 8-bit integers. This type definition makes it possible to hold any arbitrary amount of data that flows between a network and a program.

Buffer API also provides a long set of primitives for data restructuring, conversion, and transformation, including conversions to and from strings, a common JavaScript type.

Buffers are an inevitable use case for stream-based data transport across I/O endpoints.

There was a question in the previous section – Node.js streams: why is the data received in the data handler function 'stringified' before it is printed? What

would happen if the data is printed as is? The answer is obvious by now. The type of the data is Buffer not string, hence the 'stringification'. If we print the data as is, we will get a 'stringified' version of the buffer object, which is a few bytes of the data in its binary form. Let's see that. The following code shows reading from the file as a stream, but printing the data as is:

```
1. const f = require('fs')
2. const r = f.createReadStream(__filename)
3. r.on('data', (d) => {
4.   console.log(d)
5. })
```

And the output will be as follows if we execute that:

```
code — -bash — 42×10 — ⌥⌘2
#node buffer1.js
<Buffer 63 6f 6e 73 74 20 66 20 3d 20 72 6
5 71 75 69 72 65 28 27 66 73 27 29 0a 63 6
f 6e 73 74 20 72 20 3d 20 66 2e 63 72 65 6
1 74 65 52 65 61 64 53 74 72 65 ... 57 mor
e bytes>
#
```

Figure 5.14: *Output with a sample Buffer program*

Evidently, the data that comes in the handler is a buffer object, so the printed matter is unreadable. On the other hand, the following code shows the same scenario, but the data is converted with '**`toString()`**' before printing:

```
1. const f = require('fs')
2. const r = f.createReadStream(__filename)
3. r.on('data', (d) => {
4.   console.log(d.toString())
5. })
```

And we have the right data printed, as in the following screenshot:

```
code — -bash — 42×10 — ⌥⌘2
#node buffer2.js
const f = require('fs')
const r = f.createReadStream(__filename)
r.on('data', (d) => {
  console.log(d.toString())
})

#
```

Figure 5.15: *Output with the Buffer program*

Why do we need to perform the stringification?

- First, we used web server because of architectural considerations
- Next, we used established protocols to transfer data
- Then, we had to use stream to be efficient in network I/O
- We devised Buffer, a new generic type that can hold arbitrary data

This is why we are casting the data from a generic type to the known type at the point of its actual use.

The following are some common examples of Buffer API usages:

1. Create a **`Buffer`** object of some size and fill it with some data:

   ```
   1. const b = Buffer.alloc(4, 'ABCD')
   2. console.log(b)
   ```

   The output of the preceding code is as follows:

   ```
   code — -bash — 42×7 — ⌥⌘2
   #node buffer3.js
   <Buffer 41 42 43 44>
   #
   ```

Figure 5.16: Output of Buffer object

2. Create a Buffer object from a string:

```
1. const b = Buffer.from('node')
2. console.log(b)
```

The output is shown as follows:

```
code — -bash — 42×7 — ⌥⌘2
#node buffer4.js
<Buffer 6e 6f 64 65>
#
```

Figure 5.17: Output of Buffer holding a string

3. Cast two bytes as a 16-bit integer from a **`Buffer`** object in Little Endian mode:

```
1. const b = Buffer.from([0x0a, 0x0b])
2. console.log(b.readInt16LE(0))
```

And the output is shown as follows:

```
code — -bash — 42×7 — ⌥⌘2
#node buffer5.js
2826
```

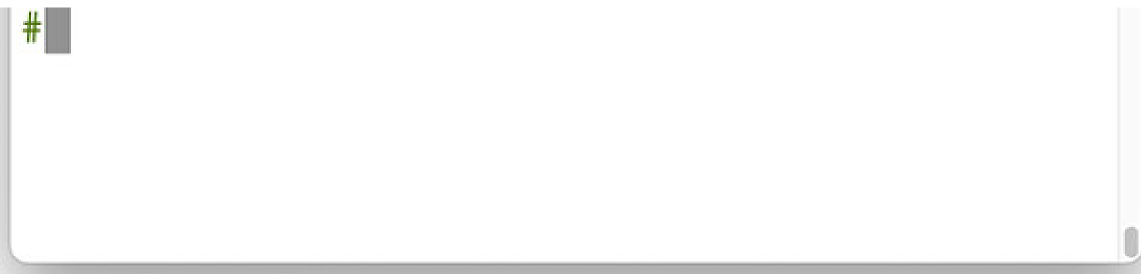

Figure 5.18: *Output of LE integer read*

What is the value 2826 here? It is the decimal equivalent of 0x0b0a. We want to cast the bytes in Little Endian mode, so the higher-order bytes will appear as the most significant ones, and vice-versa.

4. Cast two bytes as a 16-bit integer from a Buffer object in Big Endian mode:

```
1. const b = Buffer.from([0x0a, 0x0b])
2. console.log(b.readInt16BE(0))
```

And this is the output:

```
code — -bash — 42×7 — ⌥⌘2
#node buffer6.js
2571
#
```

Figure 5.19: *Output of BE integer read*

Similarly, what is the value 2571 here? It is the decimal equivalent of 0x0a0b. As we are casting in Big Endian mode, the lower-order bytes will appear as the most significant ones, and vice-versa.

In summary, Buffer provides a convenient way to offload stream data from an I/O. The class provides a lot of convenient methods to create, cast, and

transform data to and from its raw representation.

Request and response objects

So far in this chapter, we have learned that network programming requires special protocols, HTTP is a common network protocol, network data transport is carried out in Node.js through streams, and that stream data itself is largely held in Buffers.

Now, let's revisit the famous networking abstractions at the server side: the 'request' and the 'response' objects. In an HTTP-based Node.js server implementation, these objects are pre-populated and made available to a connection handler function. As we have reiterated a couple of times in the previous chapters, the 'request' object represents the client request, and the 'response' object represents the server's response.

We asked a question in the HTTP section of this chapter: When the server receives a request, it is not in the form of a standard HTTP message. Instead, the individual values are decomposed and constructed into s 'request' object. Who performs this activity?

An easy way to determine this is to trace the history of the program sequence that led to the calling of request handler callback. For that, just insert '**`console.trace()`**' in the callback and observe the call stack. The following figure shows the call stack upon entry to the request handler callback:

```
Trace
    at Server.<anonymous> (foo.js:3:11)
    at Server.emit (events.js:314:20)
    at parserOnIncoming (_http_server.js:782:12)
    at HTTPParser.parserOnHeadersComplete (_http_common.js:119:17)
#
```

Figure 5.20: *The call-stack at request handler function*

The HTTP message is first parsed by an HTTP parser. Upon validating the sanctity of the message and when the parser believes it has enough information, it populates the 'request' object with the relevant message parts and invokes the request handler callback. When is this done? At the message parsing phase. What is this action called? This action is called HTTP parsing.

Request

A request object is of '**`http.IncomingMessage`**' type. It is a readable stream, allowing it to manage flowing data while inhibiting writing into it. This is created each time when there is a client request and is made available to the client request handler function. Being a stream helps it exhibit flowing data. This is convenient in cases where the request data cannot be made available to the server atomically (for example, large data uploads). At the same time, HTTP parsing would have been fully performed by the time it reaches the handler, providing an efficient way for the programmer to start processing the request.

For that matter, the object that the request-initiating entity (usually the client) receives upon obtaining a response from the other endpoint is also of `**`http.IncomingMessage`**' type. This is not accidental. Although the nature of these two entities might look fundamentally different at first glance, closer inspection reveals similarity in these aspects:

- Both represent a message arrival (one from the client, the other from the server)
- Both represent a message in transit (readable streams)
- Both have similar life cycle events

The key difference is that the former is called a request in its spirit and purpose, while the latter is actually a response to a previous request from the remote. Do not confuse this with the '**`ServerResponse`**' object, which we will discuss later. The request object contains the underlying socket, which is a handle to the remote endpoint.

The following figure shows the data types of the three objects we mentioned earlier:

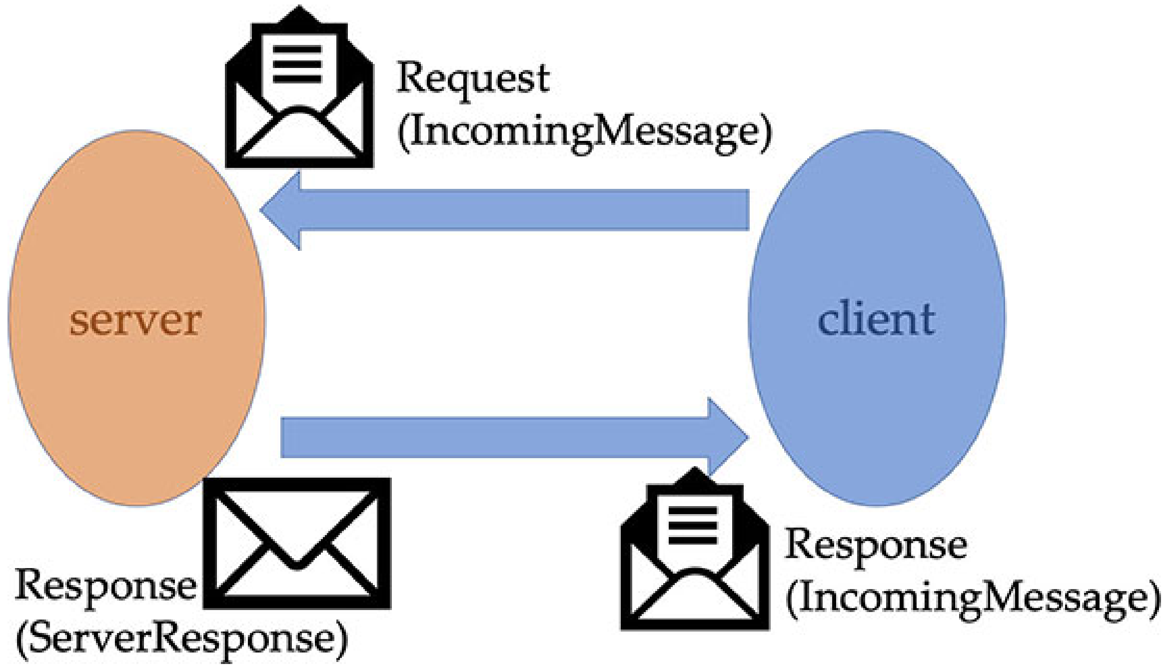

Figure 5.21: *Request and response objects and their placement*

The following code snippet shows the common usage of the request object with both the client and server in a single program so that we can see the type of request object as well:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   console.log(`In server, q is a ${q.constructor.name}`)
4.   console.log(`client said: ${q.url}`)
5.   r.end('/bar')
6. }).listen(12000, () => {
7.   h.get('http://localhost:12000/foo', (m) => {
8.     console.log(`In client, m is a ${m.constructor.name}`)
9.     m.on('data', (d) => {
10.      console.log(`server said: ${d.toString()}`)
11.    })
12.    m.on('end', () => {
13.      s.close()
14.    })
```

```
15.     })
16. })
```

Here's the output:

```
code — -bash — 36×6 — ⌥⌘2
#node request.js
In server, q is a IncomingMessage
client said: /foo
In client, m is a IncomingMessage
server said: /bar
#
```

Figure 5.22: *Output that shows the request object types*

Response

A response object is of '**`http.ServerResponse`**' type and is a stream. This is created each time there is a client request and is made available to the client request handler function, along with the request object. Being a stream helps it manage the server's response as flowing data. It is a 'WritableStream', so the response can be written to it. It is a 'ReadableStream' too, so the server program itself can read from the in-light (half-baked) response that has not been sent to the client and flushed from the send queue. Key functions of this object include writing the response header and body and controlling the data flow through the stream interface. The response object also contains the underlying socket, which is a handle to the remote endpoint. So, this object is used if we want to control the configuration of the underlying network transport pertinent to the server's response.

Question: What is the use case for reading from the response stream by the same program that wrote into it?

Here are a couple of programs that illustrate some aspects of the response objects as well as showcase general usage. The following code shows the usage of the response object to set a custom header:

```
1. const h = require('http')
```

```
2. const s = h.createServer((q, r) => {
3.   r.setHeader('hello', 'world')
4.   r.end(r.getHeader('hello'))
5. }).listen(12000, () => {
6.   h.get('http://localhost:12000', (m) => {
7.     m.on('data', (d) => {console.log(d.toString())})
8.     m.on('end', () => {s.close()})
9.   })
10. })
```

The response object is a writable stream, so we can write valid HTTP messages into it. And it is a readable stream too, so we can also read back from it. In line #4, we are reading the header that we set earlier and sending it as a message body as well, just to show that we can read back from the response. The output is shown in the following screenshot:

Figure 5.23: *Output of code with the response object*

The following code shows the use case with the '**`writeHead`**' API, which also serves a similar purpose as that of the '**`setHeader`**' API but inhibits reading back data, in the absence of a '**`setHeader`**' usage:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   r.writeHead(200, {'hello': 'world'})
4.   r.end(r.getHeader('hello'))
5. }).listen(12000, () => {
```

```
6.  h.get('http://localhost:12000', (m) => {
7.    m.on('data', (d) => {console.log(d.toString())})
8.    m.on('end', () => {s.close()})
9.  })
10. })
```

The intent is similar to the '**`setHeader`**' case, but the '**`getHeader`**' API was unable to get any data from the response object, as the headers were immediately written to the network and flushed from the program's memory! This is illustrated as follows:

```
code — -bash — 36×6 — ⌥⌘2
#node response2.js
#
```

Figure 5.24: *Output with the use of 'writeHead'*

The function '**`setHeader`**' can be used when a large number of headers are being set in an iterative manner with potential for later retrieval and modification, while '**`writeHead`**' is used to respond to the client as and when we have data, with no caching of intermediary headers in the response object. Obviously, there is no later modification possible with this approach.

Request and response life cycle

As mentioned earlier, both request and response objects are stream objects, so the life cycle of these objects is essentially that of a data stream. This helps the program install handlers for interested life cycle events and take appropriate action. The most common life cycle events are as follows:

Request life cycle

There are two discrete ways this object is used: i) as an abstraction that holds the client request in the server upon a client request, and ii) as an abstraction that holds the server response in the client upon the server's reply. Apart from the usual life cycle events of a 'ReadableStream' that apply as is, these additional events are defined:

'aborted'

This event is emitted when the underlying client request is aborted. Any subsequent processing of the request is deemed invalid. Programs can install a handler for this event to catch such situations and gracefully shutdown any subsequent actions on this request.

The following code illustrates this event being captured at the 'IncomingMessage' objects. The request is explicitly aborted here, but this could come as any network glitch in the real world:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   q.on('aborted', () => {
4.     console.log('client request aborted')
5.   })
6.   r.write('ok')
7. }).listen(12000, () => {
8.   const r = h.get('http://localhost:12000', (m) => {
9.     m.on('data', (d) => {
10.      console.log(d.toString())
11.    })
12.    m.on('end', () => {})
13.    m.on('aborted', () => {
14.      console.log('server response aborted')
15.      s.close()
16.    })
17.    r.abort()
18.  })
19. })
```

The following output shows that when the client request is aborted, the same is reflected in the server as an event, which, in turn, percolates to the client who requested it in the form of an 'aborted' event on the server's response:

```
code — -bash — 36×6 — ⌥⌘2
#node request_aborted.js
server response aborted
client request aborted
#
```

Figure 5.25: *Output of code with 'aborted'*

'close'

This event is emitted when the underlying client connection is closed. Again, programs can use this event to perform post-close activities if any, on the connection.

Apart from these, the most common events usually performed on a request object (inherited from 'ReadableStream') are:

'data'

This event is emitted when the underlying stream has data available. Install a handler to consume it. On the server side, this handler is usually required only when a client sends a large amount of data (upload use case using HTTP POST verb), as other types of requests carry small amount of data and are readily available in the respective fields of the request object. On the client side, usually a data handler is installed to 'catch' the server's response.

There are two ways of using the 'data' handler: i) those that are capable of operating on arbitrary amount of data, ii) those that buffer the data until it receives everything.

The following code shows a handler that works on an arbitrary amount of data, fully aligning to the principles of stream:

```
1. m.on('data', (d) => {
```

```
2.   // do work on d
3.   // (arbitrary type,
4.   // arbitrary length)
5. })
```

The following code shows a handler that buffers the stream data until the data flow has ended, before it starts working on it:

```
1. const data = ''
2.
3. m.on('data', (d) => {
4.   data += d
5. )}
6.
7. m.on('end', () => {
8.   // work on data
9.   // (complete data)
10. })
```

Both the cases are common, and there is no merit to one over the other. Depending on the specific scenario, you may choose whichever is convenient.

‘end’

This event is emitted when the request’s underlying stream has no more data to read, and the writing end of the stream has acknowledged the completion of writing. This is a powerful construct in the case of streams – as the data is flowing, the programs should know when the data reception is complete, precisely, and consistently.

Whether on the client or server side, programs usually start processing the request data when this event occurs, as it marks the complete availability of data.

```
1. stream.on('end', () => {
2.   // all the data has arrived
3.   // process the data gathered so far
4. })
```

Response life cycle

As the response is a full duplex stream object, it has a large set of events associated with both writable and readable streams. The ones specific to the response objects are as follows:

‘close’

This event is emitted when the response is completed or if and when the underlying connection breaks in between. Programs can install a handler for this event to catch such situations and gracefully shutdown any subsequent actions on this response.

‘finish’

This event is emitted when the response is dispatched from the server process. While this does not guarantee that the response has reached or will be reaching the client (due to various factors like network breakage), it is an indication that all the data pertinent to the server’s response has been submitted to the server system’s operating system, which, ultimately, is responsible for transporting the data. The following code illustrates an example usage of the ‘finish’ event:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   r.on('finish', () => {
4.     console.log('response finished')
5.   })
6.   r.end('hello')
7. }).listen(12000, () => {
8.   h.get('http://localhost:12000', (m) => {
9.     m.on('data', (d) => {})
10.    m.on('end', () => {s.close()})
11.  })
12. })
```

This event is emitted after the call to ‘end’. It may be an exercise to see which event occurs first - the ‘finish’ event in the server, or the message arrival in the client.

```
code — -bash — 36×6 — ⌥⌘2
#node response_finish.js
response finished
#
```

Figure 5.26: *Output of code with the 'finish' event*

Server configuration

So far, we have examined request and response objects, their specific roles in the server, and their life cycle events. We also know that these objects implement the 'HTTP' protocol, which also leverages TCP/IP underneath, and that all data transport is managed as streams. Now, let's look at the server object itself, which abstracts a web server and manages the client communication through a system-defined abstraction called **socket**.

The server object is created by a call to '`createServer`' of the '`http`' module. This object is inherited from a TCP server, aligning to what we learned earlier —HTTP communication uses TCP communication underneath. The following figure shows a live dump of the relevant fields of a server object:

```
01.  Server {
02.    _eventsCount: 2,
03.    _connections: 1,
04.    _handle: TCP {
05.      reading: false,
06.      onconnection: [Function: onconnection],
07.      [Symbol(owner_symbol)]: [Circular *1]
08.    },
09.    allowHalfOpen: true,
10.    pauseOnConnect: false,
11.    httpAllowHalfOpen: false,
12.    timeout: 0,
13.    keepAliveTimeout: 5000,
14.    maxHeadersCount: null,
15.    headersTimeout: 60000,
16.  };
```

Figure 5.27: *A sample server object dump*

The server object is used to control certain server characteristics that affect all the connections that the server handles. This includes parameters pertinent to data flow, connection inactivity, and maximum concurrent connections. These can be modified at any point in time – at the time of creating the server, after the creation, or even after handling some connections.

'maxHeadersCount'

This parameter defines the maximum number of incoming headers per request. 0 means no limit is imposed.

In the following code, the server starts with `maxHeadersCount` configurable (user passes the actual value in the command line) and the client adds three headers to its request:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   console.log(q.headers)
4.   r.end('hello')
```

```
5. })
6. s.maxHeadersCount = +process.argv[2]
7. s.listen(12000, () => {
8.   h.get('http://localhost:12000',
9.     {headers: {a: 'b', c: 'd', e: 'f'}}, (m) => {
10.    m.on('data', (d) => {console.log(d.toString())})
11.    m.on('end', ()  => {s.close()})
12.  })
13. })
```

As we can see in the following screenshot of the output, only the number of headers specified at the server configuration is allowed in any request:

```
#node server_headers.js 1
{ a: 'b' }
hello
#node server_headers.js 2
{ a: 'b', c: 'd' }
hello
#node server_headers.js 3
{ a: 'b', c: 'd', e: 'f' }
hello
#node server_headers.js 10
{
  a: 'b',
  c: 'd',
  e: 'f',
  host: 'localhost:12000',
  connection: 'close'
}
hello
#
```

Figure 5.28: *Output with the header configuration code*

‘timeout’

This parameter defines the amount of inactivity for the server sockets (that are created after new client connections) before the connection times out. It enforces the server to prevent slow backends from bloating the server with

resources. In the following code, the server receives a request from the client but responds only after 10 milliseconds. However, the server has setup an inactivity timeout of 1 millisecond, which is much lower than the current request's inactivity:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   setTimeout(() => {
4.    r.end('hello')
5.   }, 10)
6. })
7. s.timeout = 1
8. s.listen(12000, () => {
9.   const r = h.get('http://localhost:12000', (m) => {
10.    m.on('data', (d) => {console.log(d.toString())})
11.    m.on('end', ()  => {s.close()})
12.  })
13.  r.on('error', (e) => {
14.    console.log(e)
15.    s.close()
16.  })
17. })
```

As a result, the underlying connection gets reset, and the request does not complete, as shown in the exception here:

```
code — -bash — 77×14 — ⌥⌘2
#node server_timeout.js
Error: socket hang up
    at connResetException (internal/errors.js:612:14)
    at Socket.socketOnEnd (_http_client.js:493:23)
    at Socket.emit (events.js:326:22)
    at endReadableNT (_stream_readable.js:1244:12)
    at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  code: 'ECONNRESET'
}
#
```

Figure 5.29: Output with the code that changes 'timeout'

Server's life cycle events

A Node.js HTTP server is also a TCP server instance. It is also an '**`EventEmitter`**' instance to be able to manifest event-driven behavior. The server object defines a set of events that cover its life cycle. Remember that the life cycle of a connection and a server are subtly different. The former represents a specific connection with a client, while the latter represents the whole server itself, which has a wider life span. Some important life cycle events are listed here:

'listening'

This event is emitted when the server starts listening and is ready to accept connections. It is a onetime event in the server's life.

The following examples illustrate the importance of this event:

In the first example, the server listens at port 12000, and then we start the client that connects to the server and makes a request:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3. r.end('hello')
4. })
5. s.listen(12000)
6. const r = h.get('http://localhost:12000', (m) => {
7.   m.on('data', (d) => {console.log(d.toString())})
8.   m.on('end', ()  => {s.close()})
9. })
```

And what happens? Do you see anything evidently wrong here?

```
#node server_listening1.js
events.js:291
      throw er; // Unhandled 'error' event
      ^

Error: connect ECONNREFUSED 127.0.0.1:12000
    at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1144:16)
Emitted 'error' event on ClientRequest instance at:
    at Socket.socketErrorListener (_http_client.js:469:9)
    at Socket.emit (events.js:314:20)
    at emitErrorNT (internal/streams/destroy.js:100:8)
    at emitErrorCloseNT (internal/streams/destroy.js:68:3)
    at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  errno: -61,
  code: 'ECONNREFUSED',
  syscall: 'connect',
  address: '127.0.0.1',
  port: 12000
}
#
```

Figure 5.30: *Output with improper listen call*

The 'connection refused' message highly resembles a scenario wherein the server is not listening at that port. And the actual reason is the same! We are calling an asynchronous API 'listen' but not waiting for the actual 'listen' to complete before we start our client!

So, in our second example, we start the client only in our handler function of the '**`listening`**' event. We will invoke the listen API only after setting up the handler:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3. r.end('hello')
4. })
5. s.on('listening', () => {
6.   const r = h.get('http://localhost:12000', (m) => {
7.     m.on('data', (d) => {console.log(d.toString())})
8.     m.on('end', ()  => {s.close()})
9.   })
10. })
11. s.listen(12000)
```

And this time it works perfectly (output not shown as we have seen this pattern several times before). But as a matter of practice, the async call to listen itself gives us a completion handler, so we can use that short-cut form, as shown here:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3. r.end('hello')
4. })
5. s.listen(12000, () => {
6.   const r = h.get('http://localhost:12000', (m) => {
7.     m.on('data', (d) => {console.log(d.toString())})
8.     m.on('end', ()  => {s.close()})
9.   })
10. })
```

‘connect’

This event is emitted when a new connection is made to this server. A natural thought would be, when and where would we use this event? And what is the relation between a ‘connect’ event and the client connection callback that is triggered whenever there is a new connection?

The answer is: A ‘connect’ event is actually triggered on the server, which eventually leads up to the invocation of the client request handler, duly populating the request and response object. So, if you want to override the default behavior of how client connections are handled, this is the event you would be registering for handling.

‘close’

This event is emitted when the server shuts down. The following example illustrates this:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.    r.end('hello')
4. })
```

```
5. s.listen(12000, () => {
6.   h.get('http://localhost:12000', (m) => {
7.     m.on('data', (d) => {console.log(d.toString())})
8.     m.on('end', ()  => {s.close()})
9.   })
10.  s.on('close', () => {
11.    console.log('server closes')
12.  })
13. })
```

And the output of the program is as shown here:

```
code — -bash — 41x7 — ⌥⌘2
#node server_close.js
hello
server closes
#
```

Figure 5.31: *Output with server 'close' event*

Note: \`process.argv\` is a convenient way to tap the user input from the command line. It is an array. argv[0] is the 'node' executable itself, while argv[1] is the name of the script file. The rest of the command line arguments, if any, are made available in argv[2], argv[3], and so on. If the argument is a number, we usually precede it with a '+' sign to force its type, lest the value can get operated with other operands in strange ways.

Other networking APIs

In this section, we will quickly go through other important networking APIs that Node.js exposes. We will not be using them in this book, but this chapter will not be complete if we don't touch base on that as they are vital parts of Node.js networking capabilities.

HTTPS

This is an extension to HTTP, and the 'S' stands for secure. It provides a security layer on top of the base protocol by defining data securing semantics at the transport layer. This will imply additional hand-shaking semantics at the application level to enable secure communication. The 'https' module provides the necessary APIs for supporting applications to communicate securely.

HTTP 2

This is an enhancement to the HTTP/1.1 protocol. The protocol is around 30 years old and has some inefficiencies with respect to the premise of internetworking. HTTP 2 provides an optimized protocol specification as compared to its previous counterpart. The two key enhancements are:

- Performance optimization around key protocol design
- Ability of the server to push data to the client

Node.js provides a comprehensive set of methods to working with this protocol through the 'http2' module.

UDP

User Datagram Protocol (**UDP**) is another protocol at peer-level with TCP. This is a connection-less protocol in that it does not require the endpoints to have a prior handshake establishment for transferring data. The '`dgram`' module provides APIs that fully support the UDP protocol.

Conclusion

We have taken a full tour of the Node.js networking APIs. We thoroughly examined the need for protocols for inter-network communication among programs. We took TCP/IP and HTTP as exemplary case studies and discussed them at length. We understood why streams and buffers are so important in Node.js programs that deal with heavy I/O. We then dissected the Node.js server abstractions—'request' and 'response'—that gave us a good insight into server middleware concepts. We also examined the configurations that affect the server's behavior as well as the server's life cycle control points and server events. This enabled us to write common server programs that are functionally correct and cover a basic set of communication.

In the next chapter, we will start talking about the building blocks of a website that our server program aims to develop. We will cover static and dynamic content, routes and endpoints, and request types. We will also cover advanced topics like cookies, sessions, and request forwarding.

CHAPTER 6
Major Web Server Components

We have established and familiarized ourselves with the natural flow of this book: a bottom-up approach, wherein the bottommost layer is the workload characteristics, the implication of computer organization to those characteristics, and the placement of Node.js architecture in it. The topmost layer is the website application and its user interface. The chapters in between deal with topics that constitute the middle layers, with an increasing level of abstractions around business logic. In the previous chapter, we got good an insights into Node.js APIs on network programming. This chapter introduces the next layer—web server middleware—components that a typical web server uses and reuses for its most common use cases. These include static and dynamic content serving, routes and endpoints, HTTP verbs, request forwarding, cookies and sessions, and so on. Not every web server will need these primitives, but the concepts are generic in nature, so they constitute a reasonable, coherent set for learning together.

The following diagram depicts the progressive buildup of the web server with incremental capabilities added:

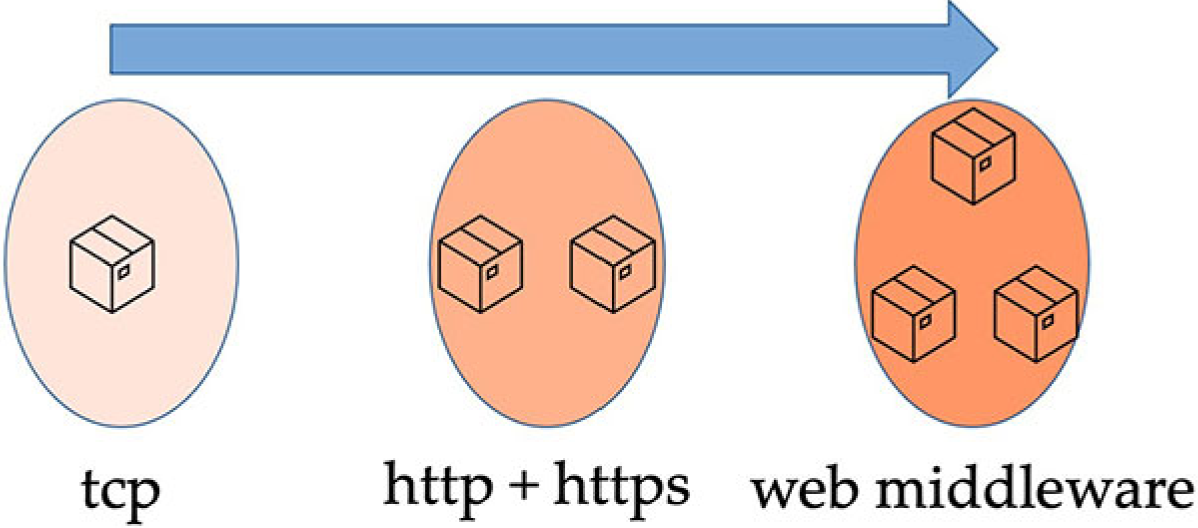

Figure 6.1: *Incremental buildup of server capabilities*

Note: Here's a simple example to help is understand the importance of a network protocol. At the lowest level, when two-byte data (B1, B2) reaches an endpoint, the question as to how to combine the bytes to reconstruct the original word—B1B2 or B2B1—is defined by a concept called endian-ness. Two types of endian-ness are prevalent: big endian and little endian.

Tip: How do we see an HTTP message header? A simple technique is to write and run a TCP server that just prints the incoming request, accessing the server through a browser client as if it is an HTTP-based server. The client will fail, but that is fine; the message we obtained in the TCP server will be the complete HTTP message that the browser has sent.

There exist specialized and reusable 'npm' modules or frameworks for all these primitives, but we will not refer to any of those. Instead, we will inspect the concept in great detail with the pertinent use case, definition, and implementation considerations - to better align with the objective of this book of developing web application with only Node.js CLI as the single tool. We also want readers to be able to re-architecture these components to fit their use cases and optimize their specific workload scenarios. The detailed study of these concepts is essential for achieving such insights and abilities.

Structure

In this chapter, we will cover the following topics:

- A static file server
- HTTP methods (verbs)
- Routes and endpoints
- Cookies
- Sessions
- Request forwarding
- Multipart form-data
- Body parser

- Cross-Origin Request Sharing (CORS)
- HTTP response codes
- A dynamic web server
- Server security

Objective

After studying this chapter, you will be able to understand the common middleware abstractions in a web server. This will include the concepts of static and dynamic content serving that define the nature of the server. You will also learn about routes and end points that help the client and server categorize request types. Further, the chapter includes illustrations of important topics like HTTP verbs (methods), and you will learn about the most popular methods (GET and POST) in great detail. Then, we will look at how to forward a client request to another server using cookies and sessions for managing client sessions. You will also understand common security issues and ways to address those. For all these components, we also will lay out the most common production issues and problem determination steps.

Introduction

To better present the server middleware theme, we start with a scratch web server with no capability and zero reusability and ask a series of connected questions to ourselves. The answers to these questions will help us progress in this chapter, and our server builds up its efficiency and reusability along the way. For each topic, we illustrate the common use case, provide the definition of the component, and spend some time discussing the design considerations around the component's implementation. That will help us better scope the component, understand various hidden features in the Web, and identify the scope of enhancements, extensions, and customizations.

To start with, how do we serve simple web content to our clients?

A static file server

We learned about a time fetching server in *Chapter 3, Introduction to Web Server*. That was one of the most trivial use cases for a web server. Another

minor use of a web server is a file server. A static file serving server can be used for describing a web server's architecture. Let's examine its purpose, meaning, and implementation.

Use case

A file server is a server system that is centrally located in a network and serves different types of file resources to its clients. The file serving can be qualified with additional attributes like:

i. with or without authentication (needing a registered user profile to access the resources)
ii. with or without navigation (the ability to view the file listings and the directory structure in the remote server)
iii. with or without type inference etc. (ability for the client to detect the type of the incoming file and assign appropriate extensions to it.)

Definition

A static file server is a server that serves static files to its clients. The files are neither fetched from another server nor generated on the fly; they reside in the server's file system. The files are usually HTML, CSS, JavaScript, or image files, but there are no restrictions on the file types. These files either directly contribute to the pages that constitute the website or are consumed by the client for different purposes.

A simple static file server architecture diagram is as follows:

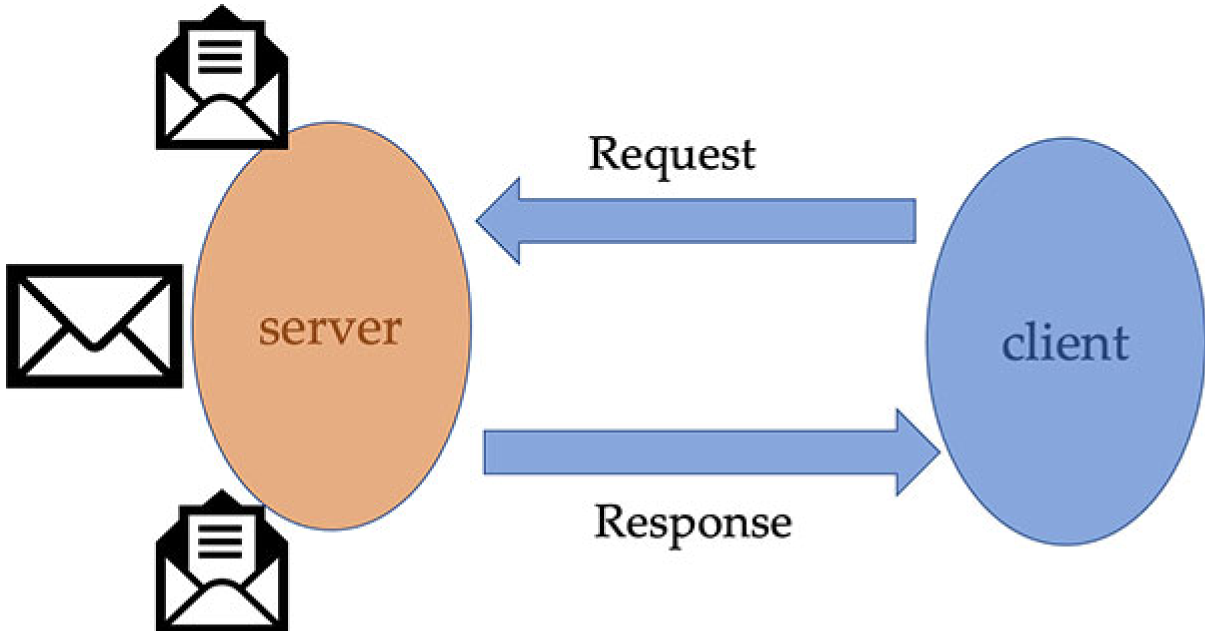

Figure 6.2: *Static file serving components*

Implementation

We will start with a trivial implementation: How to serve a file? Assuming that the file is stored in a single folder/directory in server's file system, all what we need is a way to receive the identifier of the file–mostly the filename. So, let the client request the name of the file, and the server will respond with the file content. How can we implement such a trivial file server?

- Set up a trivial server
- Read the name of the file in the request handler
- Read the file content
- Respond with the file content

The following code shows a trivial file server:

```
1. const h = require('http')
2. const f = require('fs')
3. const s = h.createServer((q, r) => {
4.   const file = q.url.substring(1)
5.   const data = f.readFileSync(file)
```

```
6.    r.end(data.toString())
7. })
8. s.listen(12000)
```

The code is trivial, except line 4. What are we doing there? 'q.url' provides the path/route/'url' that the user requests. This is usually the portion of data that is appended with the base 'url' (protocol string followed by the hostname followed by the port number if non-default is used). We use the 'substring' call to trim off the first character from the 'url', which is a '/'. When accessed through the browser, we see the following:

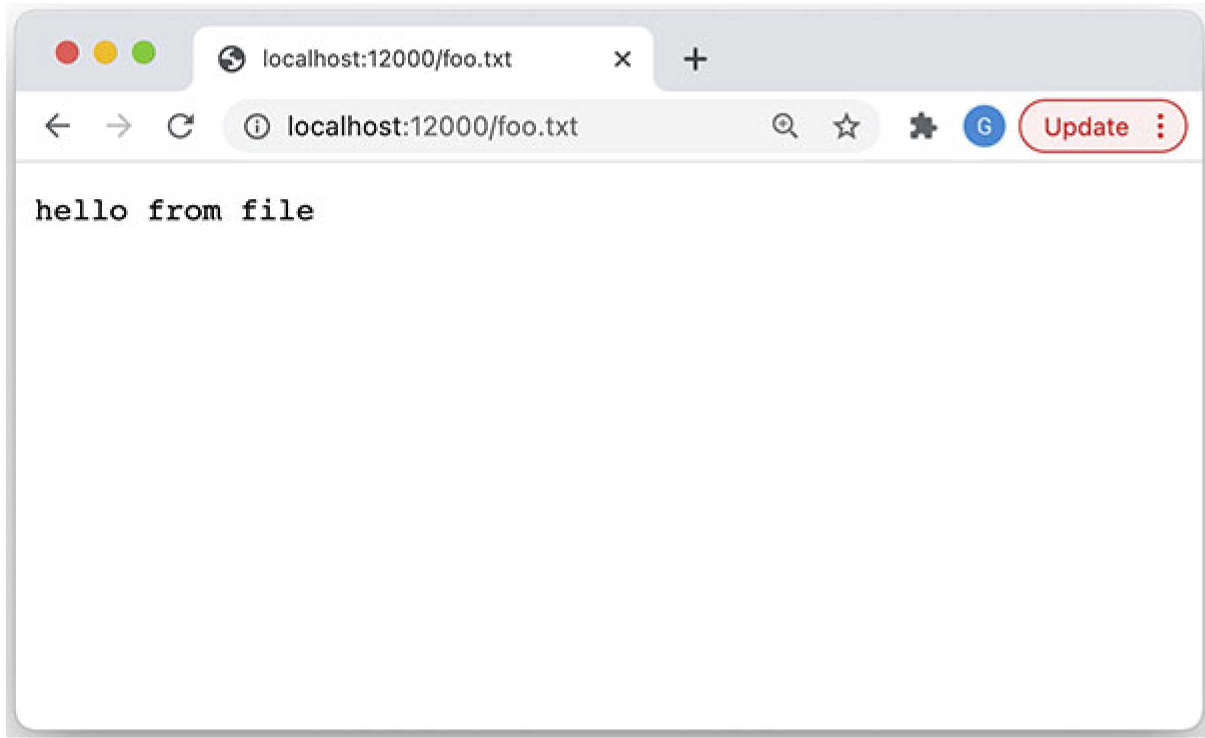

Figure 6.3: *Accessing a file serving server with predefined file location*

The first limitation here is that all the static content has to be at a single location. What if the files are organized in a typical tree structure? We have to search the file from the vortex/root of the tree structure, or we should receive the complete path of the file in question.

The first approach has a limitation in that it requires the filenames to be unique, or else there will be multiple files in the search result, causing confusion as to which one to send. So, an easy approach is to force the

requester to provide the full path so that the server code can be restructured, as shown in the following code:

```
1. const h = require('http')
2. const f = require('fs')
3. const s = h.createServer((q, r) => {
4.   const data = f.readFileSync(q.url)
5.   r.end(data.toString())
6. })
7. s.listen(12000)
```

Here's how we can consume it in the browser; pay special attention to the URL:

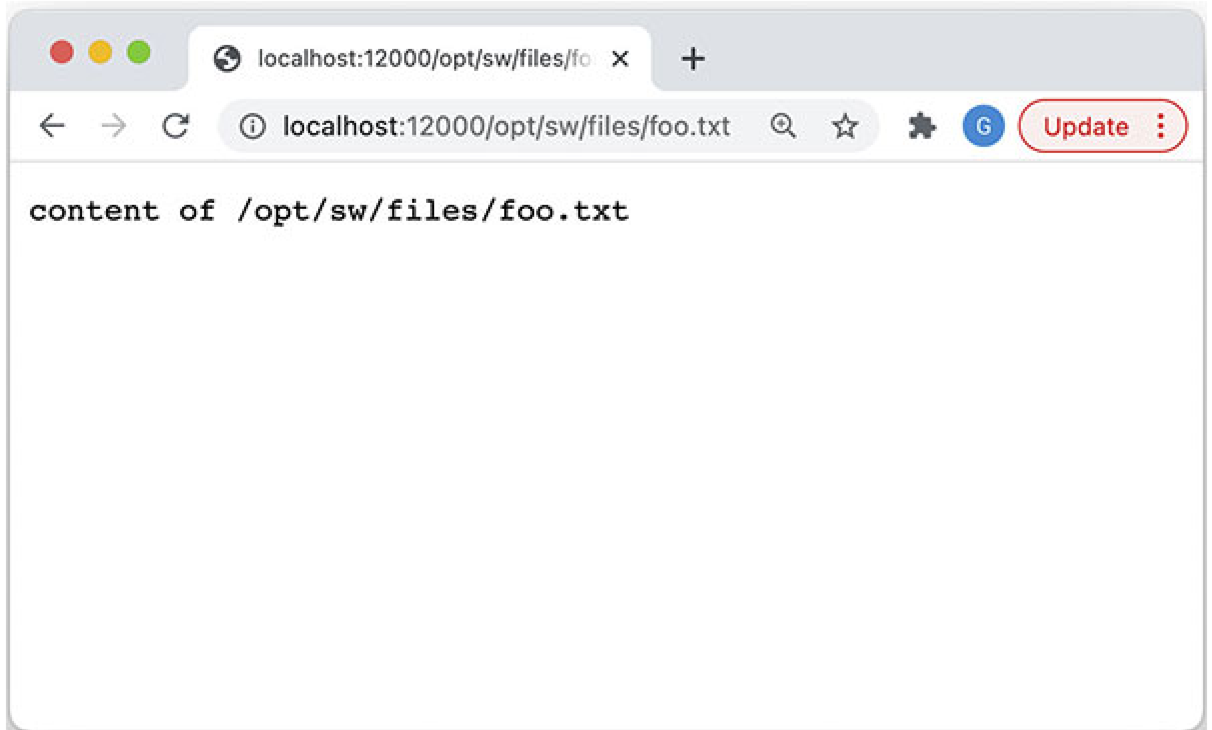

Figure 6.4: *Accessing file serving server with file location supplied by user*

A new limitation arises with this approach—the end user should know the complete path of each file! This can be alleviated by displaying the server's file structure in the browser and letting the client navigate through it rather than typing individual filenames.

A second limitation with this approach is the file type. You are sending the file content but not the type information, so how does the client know what type of file is coming? This can be addressed by setting up MIME type in the response. The following server program sets the MIME type of the file for the client's benefit:

```
1. const h = require('http')
2. const f = require('fs')
3. const s = h.createServer((q, r) => {
4.   const file = q.url.substring(1)
5.   const data = f.readFileSync(file)
6.   r.setHeader("Content-Type", 'application/zip')
7.   r.end(data)
8. })
9. s.listen(12000)
```

As can be seen, the following client downloads the compressed file as opposed to rendering its binary content on to its client area. This is the result of specifying the MIME type:

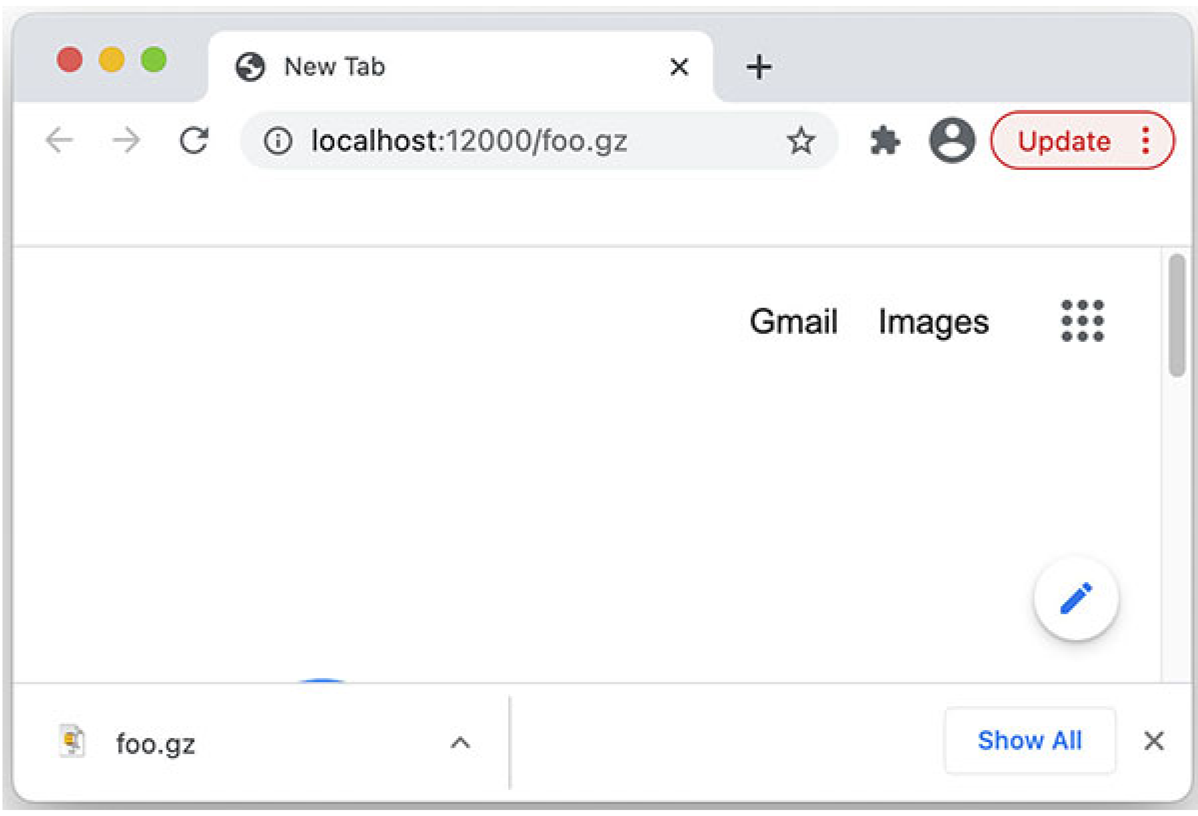

Figure 6.5: *Accessing a file serving server that is MIME aware*

Note: The Multipurpose Internet Mail Extensions (MIME) type is a standard that specifies the type and format of web content. Browsers use this information to decide what to do with the server data: render in its client area, download the file, or play as multimedia, and so on.

A third limitation with this approach is the visibility. We are allowing the client to specify an arbitrary location in the server and the server is bound to read the content as directed by the client and send it back, so we are essentially exposing the server's system details, which is a security risk.

As a solution, we can confine all requests to a specific 'data' location within the server system. Also, we need to prohibit the request filename referencing anything outside of this 'data' location, usually with the help of relative path referencing, such as '.' and '../'.

This brings the concept of 'content root'—a directory or folder that is the vortex of a tree structure scoped for the file serving needs of client requests. All file references should treat this content root as the virtual root of the file

system. In other words, a '/' in the client request should be mapped to the 'content root' in the server.

The following program illustrates the usage of the content root:

```
1. const h = require('http')
2. const f = require('fs')
3. const cr = '/var/www/website1/public'
4. const s = h.createServer((q, r) => {
5.   const file = cr + q.url
6.   const data = f.readFileSync(file)
7.   r.end(data)
8. })
9. s.listen(12000)
```

By doing this, we are only exposing a specific location within the server to the client requests, and the rest of the server file system is immune to client access.

And the following screenshot shows its usage by the client browser:

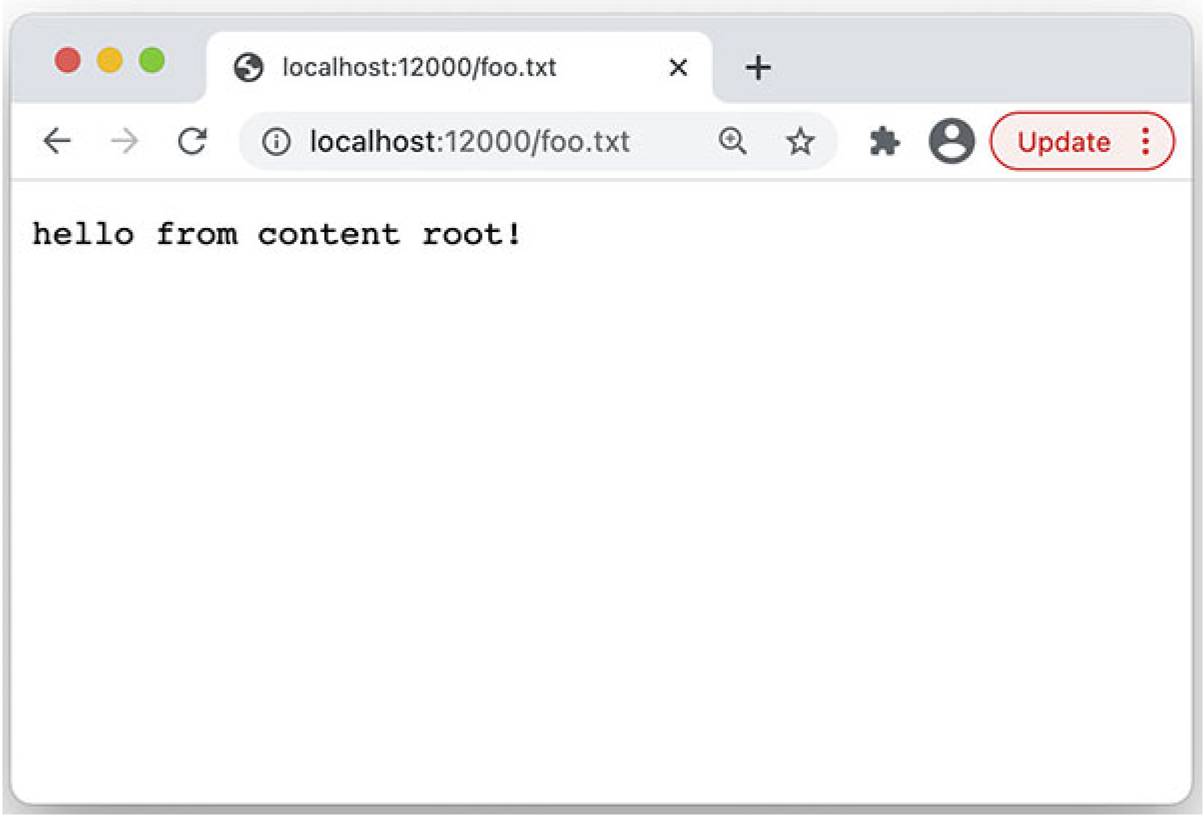

Figure 6.6: *Accessing a file serving server with content root*

Question : We are allowing the client to specify an arbitrary location in the server, and the server is bound to read the content as directed by the client and send it back, so we are essentially exposing the server's system details. This is a security risk. Why?

Other optional enhancements that we can implement on our static file server are:

- Additional file attributes can be sent (think of creation time and such)
- Define an index file for file listing (so that the client doesn't need to remember filenames)
- A way to download multiple files (for example, an entire folder as opposed to a single file)
- A way to download the whole structure as a ZIP file
- A way to cache the file content in memory to serve frequent files faster

Assignment: Extend the previous example program and accommodate the above-mentioned enhancements, one at a time. This helps you identify and use several new Node.js APIs and also improves your Node.js backend skills.

That wraps up our answer to the question of how we serve simple web content to our clients. Next, how do we make our server a little more versatile than serving static content? Can we 'manage' the files and other resources in the server as opposed to just consuming it? Can our sever be a bit more polymorphic? That is, can we allow the client to send different types of requests, intercept and interpret those request types, and serve different content based on the request?

HTTP verbs (request methods)

As our web server builds a little more capability than a static file server, the next natural thing to do is to diversify its capability. For example, can it serve two or more different types of content (though still static) for two or more different types of requests? HTTP verbs can help meet this requirement.

Use case

A typical web server implements several interfaces to abstract plurality of capabilities. A simple example is a web server that allows a user to download published images and also allows them to upload and publish newer images. This is handled by the same web server as opposed to two different servers and implemented using two different interfaces. Accordingly, the client should be able to specialize their requests to qualify further on the specific aspect of what their requests are; for example:

i. here is a request to "get" a file named foo.png
ii. I want to "upload" a file named "`bar.png`".

Definition

Request methods are the highest level of specialization of a request. These methods allow clients to indicate the purpose of the request. They allow

servers to segregate the implementation based on what purpose each interface abstracts. The verbs are designed to represent discrete, predefined actions on resources that the server abstracts, though in practice, the server is free to perform any action it deems fit.

The most used verbs are GET and POST. The GET request represents requesting the specified resource in the server, while the POST request represents requesting the creation (or updating) of a resource in the server with the data supplied in the request body.

Here's a simple figure showing an exemplary request with HTTP verbs:

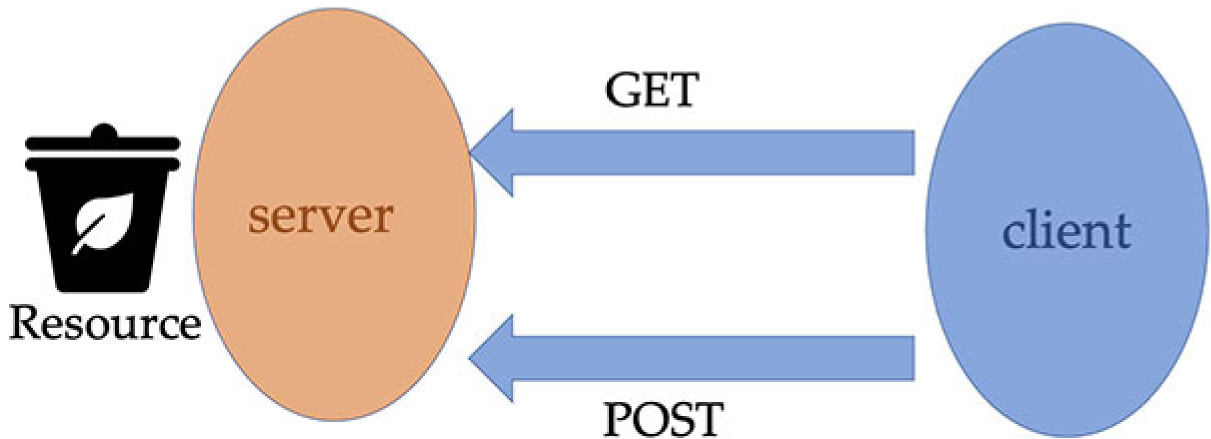

Figure 6.7: *A client-server interaction with HTTP verbs*

Implementation

On the client side, the verbs are attached to the request header, and the rest of the data is decided based on the verb. On the server side, the verbs appear in the request body. So, parsing the verb first would be a good idea to understand the request properly. Furthermore, the server can perform a functional segregation based on the request type. The most trivial segregation can be a switch case with the variable that holds the verb as the switch variable and its possible values as the case headers.

The following code illustrates this model:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   switch(q.method) {
4.     case 'GET':
```

```
5.        r.end('GET invoked')
6.        break;
7.      case 'POST':
8.        r.end('POST invoked')
9.        break;
10.     default:
11.       r.end('default response!')
12.       break;
13.   }
14. })
15. s.listen(12000)
```

With that in place, clients can send their intent by specializing the request. For example, a typical ‘**`GET`**’ request will look like this:

```
1. const h = require('http')
2. const r = h.get('http://localhost:12000', (m) => {
3.   m.on('data', (d) => {
4.     console.log(d.toString())
5.   })
6. })
```

Similarly, the following is a client that makes a **`POST`** request:

```
1. const h = require('http')
2.
3. const r = h.request({port: 12000, method: 'POST'}, (m) => {
4.   m.on('data', (d) => {
5.     console.log(d.toString())
6.   })
7. })
8. r.end()
```

A convenient optimization that can be performed at the server is to define handlers with well-known verbs as the name and attach those handlers to the server. This way, the segregator can easily invoke the appropriate handler based on the request method type.

The following code has specialized handlers based on request types:

```
1. const h = require('http')
2.
3. function onGet(q, r)  {
4.   r.end('GET invoked')
5. }
6. function onPost(q, r)  {
7.   r.end('POST invoked')
8. }
9.
10. const s = h.createServer((q, r) => {
11.   if (q.method == 'GET')
12.     onGet(q, r)
13.   else if (q.method == 'POST')
14.     onPost(q, r)
15.   else
16.     r.end('default response!')
17. })
18. s.listen(12000)
```

The HTTP verbs and their well-known meanings are as follows:

Verb	Meaning
GET	Get a resource from the server (to the client)
POST	Set a resource in the server with the client-supplied one
PUT	Replace a resource in the server with the client-supplied one
DELETE	Delete a resource in the server

HEAD	Get a status from the server (to the client)
CONNECT	Establish a tunnel between the client and server
TRACE	Perform a message loop back test with the server
OPTIONS	List the communication options exposed by the server

Table 6.1: *List of HTTP verbs*

"GET" and "POST" are the most common verbs among these.

Implement row-level security that restricts the Salespersons to view the data only for their assigned reg Question: Getting back to our static file serving server, what would be a reasonable use case for the above-mentioned HTTP verbs? That is, what kind of verbs can be used to enhance/extend the file serving server? **ions.**

If your web server is designed to manage many such request types, one of the important design considerations is to 'hide' the request parsing logic in the layer beneath and then implement handlers that are specialized to perform specific request types. The bottom layer focusses on intercepting the allowed request types, dispatching appropriate handlers, handling exceptions in case of inappropriate request types, and so on.

Note: Given that the modern workload is moving away from monolith and embracing microservice architecture, bear in mind that too many request types being handled in a single server can make it grow toward a monolith, losing the characteristic traits of microservices.

With that, we call our server reasonably polymorphic. But is that all? How do we represent different resources in our request? That is, how do we allow the client to send a request while also being able to provide path specification for different resources as well as small tokens of data that enriches the request?

Route

Building further on the versatility of our server based on HTTP verbs, routes overload the requests with different parameters to make it (the

server) reusable even further. This makes the server truly polymorphic, as it can now handle a wide variety of clients.

Use case

As we mentioned in the HTTP verb case, a server implements several interfaces to abstract the plurality of capabilities. Furthermore, the server should allow means to specify the plurality of resources that it has. A simple example is a web server that allows a user to download published images by specifying the exact path of the filename in the request. Further, a client should be able to provide a relative path of the filename. It should also specify parameterized path specifiers. The server should follow the way in which the path was specified and translate the request accordingly.

Definition

Routes are the next level of specialization of a request, after the request method. It allows clients to specify the resource required in the request. The route is part of a client request semantics, specifying a target resource. A URI reference represents the target resource, but the reference can be overloaded with additional qualifiers through a mutually understood protocol, for covering a general class of resource referencing requirements.

The following diagram illustrates a client-server interaction with the routes highlighted:

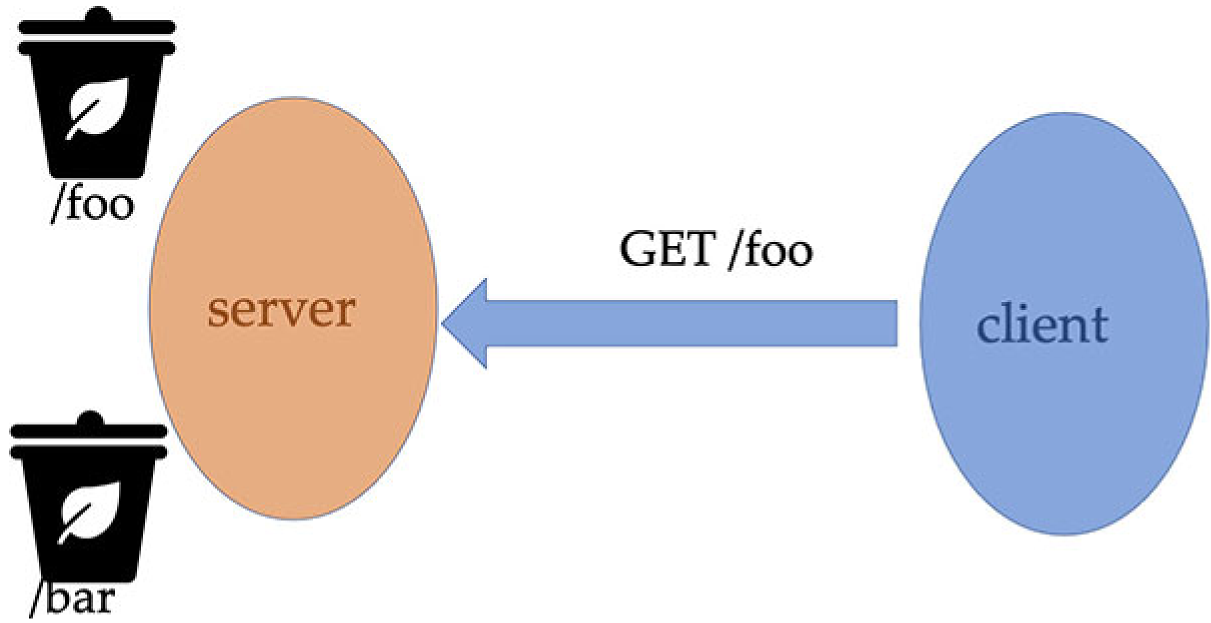

Figure 6.8: *A client-server interaction that uses routes*

Implementation

Just like the request method, on the client side, the routing path is attached to the request header and is largely referred to as path. On the server side, the route appears in the request body. Again, as in the case of verbs, parsing the route first would be a good idea to understand the request in its fullest. If there are only a finite number of routes available for the specific application, the server can perform a functional segregation based on the route. The most trivial segregation can be a switch case with the variable that holds the route as the switch variable and its possible values as the case headers.

The following code shows a simple route handling:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   switch(q.url) {
4.     case '/foo':
5.       r.end('foo path invoked')
6.       break;
7.     case '/bar':
```

```
8.         r.end('bar path invoked')
9.         break;
10.      default:
11.        r.end('default response!')
12.        break;
13.   }
14. })
15. s.listen(12000)
```

And a client request can be specialized by adding path specifiers to the base URL, as shown here. The following client request contains a '**`/foo`**' route:

```
1. const h = require('http')
2. const r = h.get('http://localhost:12000/foo', (m) => {
3.   m.on('data', (d) => {
4.     console.log(d.toString())
5.   })
6. })
```

And the next one contains a '/bar' route:

```
1. const h = require('http')
2. const r = h.get('http://localhost:12000/bar', (m) => {
3.   m.on('data', (d) => {
4.     console.log(d.toString())
5.   })
6. })
```

On the other hand, having a generic handler function that handles the routes dynamically is more suitable if infinite routes are possible (such as numerous static filenames).

A clean implementation will look like this:

```
1. const h = require('http')
2. function onFoo(q, r)  {
```

```
3.   r.end('foo invoked')
4. }
5. function onBar(q, r)  {
6.   r.end('bar invoked')
7. }
8. const s = h.createServer((q, r) => {
9.   if (q.url == '/foo')
10.    onFoo(q, r)
11.  else if (q.url == '/bar')
12.    onBar(q, r)
13.  else
14.    r.end('default response!')
15. })
16. s.listen(12000)
```

Note: A word on the name selection of routes: While there is no standard for what would make up a great route name, and there can be unique names, having meaningful names that truly reflect what the underlying handler for the route is performing would be a best practice for both code readability and maintainability.

Endpoints

Endpoints are the combination of verbs and routes. Together, they provide a way to represent a unique interface of the server for the client to communicate to. In other words, an endpoint represents a combination of a request method and a route that constitute a meaningful request from the client's perspective.

A simple server that prints back the endpoint that the client accessed is as follows:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   r.end(`${q.method} http://localhost:12000${q.url}`)
```

```
4. })
5. s.listen(12000)
```

And a matching client code is as follows:

```
1. const h = require('http')
2.
3. const r = h.request({path: '/foo',
4.                      port: 12000,
5.                      method: 'POST'}, (m) => {
6.   m.on('data', (d) => {
7.     console.log(d.toString())
8.   })
9. })
10. r.end()
```

> **Note:** The term endpoint is overloaded, just like the word virtual machine. In many situations, an endpoint is defined as a remote computing device that can communicate with external systems through a network. Here, the endpoint simply means the request type, host, port and the URL combined together according to the protocol specification.

Now we have a server capable of consistently serving a client in multiple ways based on the nature and intent of the request. This means a client can use the server features in different ways. This implies that a client, based on the website's implemented function, can make multiple requests to the same server at different points in time. Currently, the interactions are deemed idempotent (making multiple client requests, leading the server to take the same action, and producing the same result). On the other hand, the server is not capable of keeping track of clients and client requests and making correlations. How does the server 'remember' a previous request from a client and act/optimize its functionality accordingly?

Cookie

Should the server be indifferent to clients that connect for the first time and clients that connected with it one or more times in the past? Is there any benefit of the server 'remembering' the previous visits? For example, are any code or/and data optimizations possible? Cookie addresses this aspect to make the server more intelligent.

Use case

A web server needs to know if and when a client revisits the website it hosts. Further, it would also want to 'remember' some contextual information between the visits so that the current request can be handled better with the information on the previous one.

Definition

Cookie is data that the server defines for representing the context of a client visit. It is a size-limited key-value pair. The data is first composed by the server and sent to the client as part of the response header. The client retrieves the cookie from its response header and sends it in its subsequent requests in the request header for a matching endpoint. When the server receives the request, the cookie that arrives in its header helps the server recollect the context of this client's previous request.

A simple data flow with cookie is demonstrated as follows:

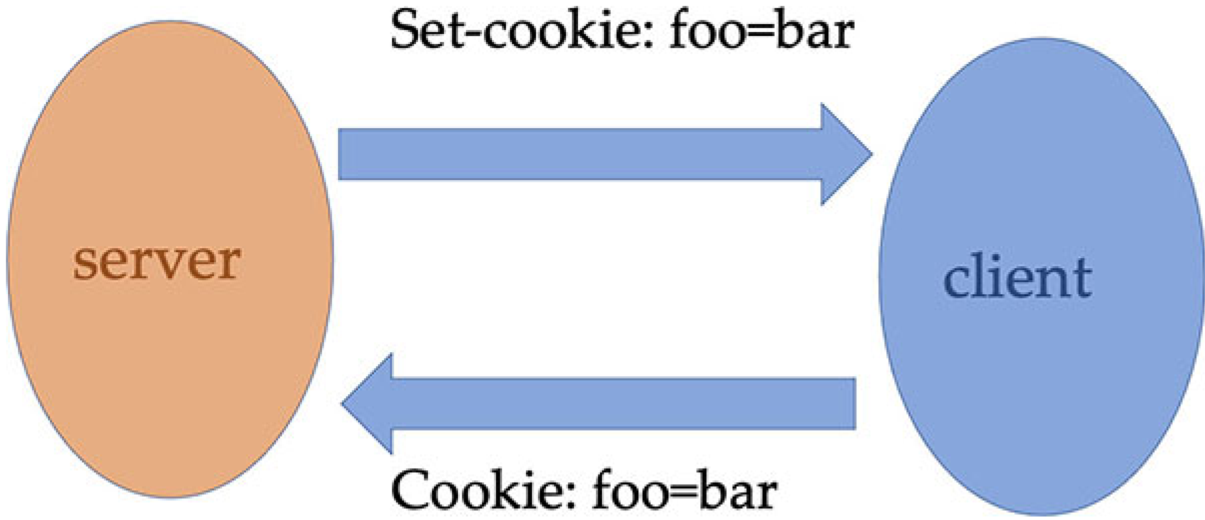

Figure 6.9: *A client-server interaction with cookies*

Implementation

The most trivial implementation has a semantic that specifies how to define a cookie at the server side and send it to the client, and how the client uses it in subsequent requests to the server.

One notable specialty of the semantics is that the attributes of a cookie are separated by semicolons, while multiple cookies are separated by commas.

There are two cookies in the preceding example represented in *figure 6.9*:

- A cookie named foo with the value hello, an expiry date, and an attribute.
- A cookie named bar with the value world.

As we can see, Boolean attributes with value true does not need to specify that as such.

The following is a piece of server-side code that sends a cookie in its response to the client while retrieving it in subsequent requests from the client to associate the requests:

```
1. const h = require('http');
2.
3. const c = ['sessionkey=fd0ea63; \
4.             expires=Sun, 1-Jan-2100 12:00:00 GMT']
5. const s = h.createServer((q, r) => {
6.   console.log(q.headers['cookie'])
7.   r.setHeader('set-cookie', c)
8.   r.end('cookie sent')
9. })
10. s.listen(12000)
```

In the client, the cookie is available in the response header, under `set-cookie` head:

```
1. const h = require('http')
2. const r = h.get('http://localhost:12000', (m) => {
3.   console.log(m.headers['set-cookie'])
```

```
4.   m.on('data', (d) => {
5.     console.log(d.toString())
6.   })
7. })
```

Again, in the client, the cookie value is printed as shown here:

```
01. [
02.   'sessionkey=fd0ea63;            expires=Tue, 24-Aug-2021 12:00:00 GMT'
03. ]
```

Figure 6.10: *A cookie received by a client*

From the client side, this is obtained in the response. It adds the cookies in the header for future use. However, note that the cookies are separated by a semicolon here.

Also, the additional cookie attributes are not sent back to the server. Why? Those are for the client to consume.

If you are accessing the server through the browser, hit the same address twice and check the server log; you will see the cookie header sent back by the client for the second request.

Question: We have learned that the cookie tokens are returned by the client for the server to consume and associate the client requests. What if a client does not honor this rule? That is, it receives the cookie data from the server but makes requests to the server as usual, without returning the cookie?

Cookie attributes

Here are some most commonly used cookie attributes:

Secure: Instructs the client to use this cookie only with secure connections, as it may contain sensitive or semi-sensitive information.

Domain: All the possible domain names for which this cookie is applicable.

Path: The path in the subsequent requests for which this cookie is relevant.

Session

This is the server-side counterpart of the contextual 'memory' that we discussed in the previous section. A cookie is stored on the client side, and the corresponding contextual data is stored in the server, with the ID as the matching key between both. With both session and the cookie, the server builds the complete information required to remember a client's visit history.

Use case

Similar to the use case for cookie, a web server needs to know if and when a client revisits the website it hosts, within the scope of a single iteration, and as part of browsing session. This knowledge will help the server 'remember' some contextual information between the requests so that the current request can be handled better with the information on the previous one. The most common example is that of a banking portal, where the first request leads to a user login, and the session created at the login time is used to carry over the context in subsequent requests from the same client. The use case of session is the same as that of a cookie, except that the server does not want to fully '*trust*' the client for its request management.

Definition

Session is temporary data that is used to manage the conversational state between a server and a client. Session is relevant for websites that contain multiple requests in a transaction. A session is a state information obtained in the first request and used to associate subsequent requests under one logical unit. The life cycle operations of the session are managed by the server.

The following figure illustrates a session data flow with a session store:

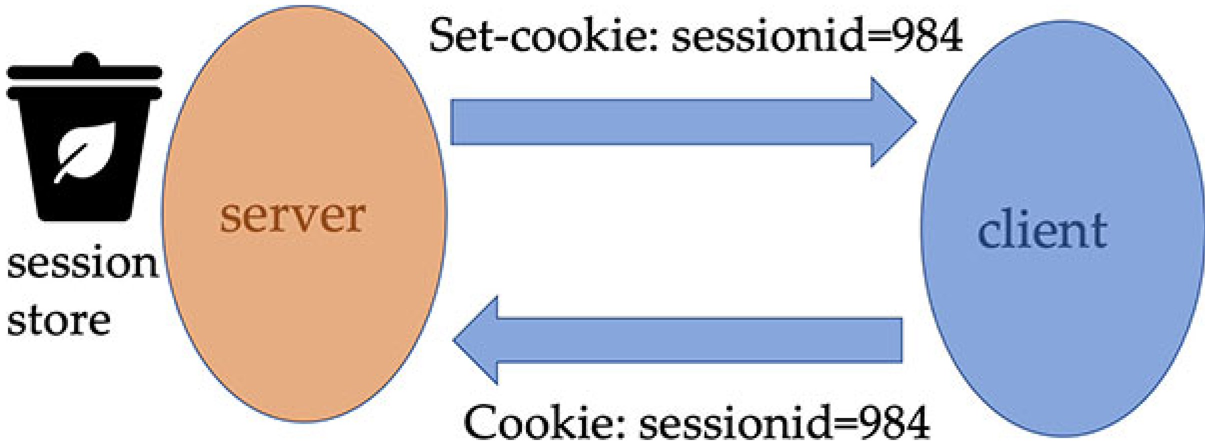

Figure 6.11: *A client-server interaction that uses session*

Implementation

Where do we store the session? Let's say at the server side. But then how do we recognize the first and subsequent requests coming from a client and associate them together? So, we need the session data to be stored in the client, just like in the case of cookies. But then there's an issue. A session stored in the client is subject to manipulation; for example, a malicious client can modify the session object, point it to some other user to hijack their sessions, or reuse previously used sessions in the computer.

To solve these issues, the session is created by the server on the first visit and stored in a data store at the server's side, and a key to the session data is sent to the client. The client sends the session key in all its subsequent requests for the server to make the association. The server obtains the actual session object with the help of the key that the client sent.

Server code that implements session is illustrated here:

```
1. const h = require('http')
2. const cr = require('crypto')
3. const m = new Map()
4. const s = h.createServer((q, r) => {
5.   const sd = q.headers['cookie']
6.   if (sd === undefined) {
7.     const hash = cr.createHash('md4')
8.     const sk = hash.digest('hex').substring(0, 10)
```

```
9.     const sv = 'my session data'
10.    m.set(sk, sv)
11.    r.setHeader('set-cookie', [`${sk}=${sv}`])
12.    r.end('cookie sent')
13.  } else {
14.    const k = sd.split('=')[0]
15.    r.end(`session: ${k}, ${m.get(k)}`)
16.  }
17. })
18. s.listen(12000)
```

What happens if the server crashes after the client logs in but before its subsequent requests? The session object created and stored in the server is flushed, and the new server (on the same host or a different node in the case of a clustered deployment) does not have access to the previous session information. In this case, the new server initiates a login again and starts a new session. This poses a weak user experience.

To remedy this, the session objects are typically stored in a central data store, and they are accessible to multiple server processes serving the same application. This central store is called **session store**.

In the following code, the store is centrally located, as opposed to the program:

```
1. const h = require('http')
2. const cr = require('crypto')
3. const s = h.createServer((q, r) => {
4.   const sd = q.headers['cookie']
5.   if (sd === undefined) {
6.     const hash = cr.createHash('md4')
7.     const sk = hash.digest('hex').substring(0, 10)
8.     const sv = 'my session data'
9.     // store this session to a central store
10.    r.setHeader('set-cookie', [`${sk}=${sv}`])
```

```
11.     r.end('cookie sent')
12.   } else {
13.     const k = sd.split('=')[0]
14.     // load the session from the central store
15.     r.end(`session: ${session}`)
16.   }
17. })
18. s.listen(12000)
```

What is the life span of a session object? Let's say it is infinite. What are the implications for a session with no set life span?

- A client that had a session a year ago can still come back and continue.
- A user using a public computer for browsing a sensitive website will leave the session open for subsequent users to enter.

Clearly, we need an expiry date for sessions. The following code illustrates this:

```
1. const h = require('http')
2. const cr = require('crypto')
3. const m = new Map()
4. const s = h.createServer((q, r) => {
5.   const sd = q.headers['cookie']
6.   if (sd === undefined) {
7.     const hash = cr.createHash('md4')
8.     const sk = hash.digest('hex').substring(0, 10)
9.     const sv = 'my session data'
10.    m.set(sk, sv)
11.    r.setHeader('set-cookie',
12.  [`${sk}=${sv}; expires=${new Date(Date.now() + 1000)}`])
13.    r.end('cookie sent')
14.  } else {
```

```
15.     const k = sd.split('=')[0]
16.     r.end(`session: ${k}, ${m.get(k)}`)
17.   }
18. })
19. s.listen(12000)
```

Question: In the previous examples, we used a random hash to create session keys. What if we use a simpler, ever-increasing number with a zero-based index? Which problem does the random hashing technique solve? What happens if the values are more predictable?

We have progressed quite a bit from a simple static file server. We can now handle different requests by type and path with full duplex (bi-directional) data flow as well as the ability to manage transactions (multiple requests that are coherent). Now, let's look at some of the maintenance aspects. An endpoint that was established earlier and was in use was cached/bookmarked by the user. But the server was refactored as part of the maintenance cycle, and that endpoint is not available in the server anymore. How do we handle such scenarios?

Request forwarding

When a website application is massively refactored, it is possible that some of the links to the existing resources change. Let's understand how request forwarding helps users work with the old links even after they have become obsolete.

Use case

Based on the usage of the website in the field, users might have cached/bookmarked the old link. With the changes in place, an uninformed user using the old link will trigger an HTTP 404 (page not found) error, and the user will not have information about the modified link. It would be great if one of the endpoints still accepts the old request (method and route) and transparently forwards it to the new resource.

Definition

Request forwarding is a technique by which a physically separated resource is allowed to be referenced by the client, and the real resource reference is supplied by the server upon receiving the old reference. The client subsequently re-requests with the new resource reference. The references that are subject to redirection can be that of a form, a page, or the entire website. This technique allows temporary or permanent redirection of pages or a site based on a various use cases.

The following diagram illustrates a request forwarding architecture:

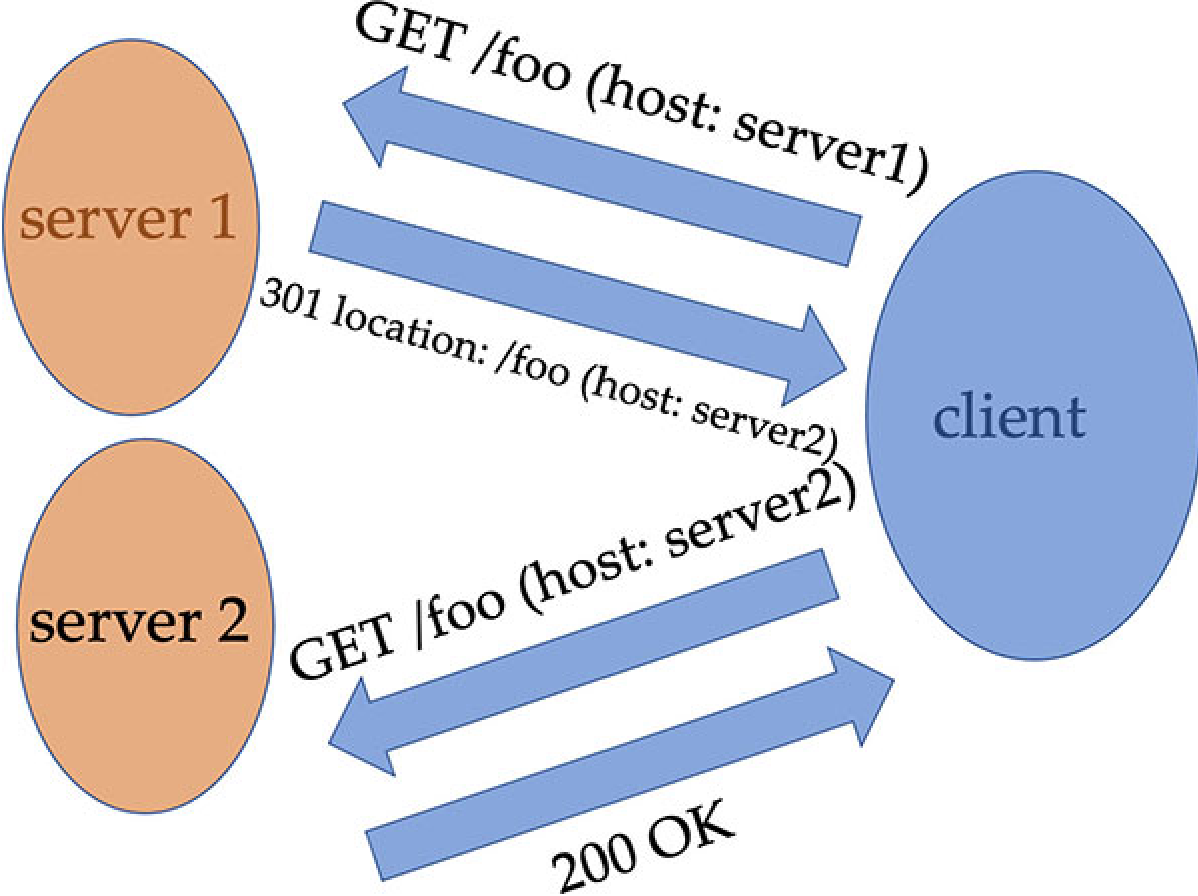

Figure 6.12: *A client-server interaction that uses request forwarding*

Implementation

The simplest way to perform redirection is to:

- Intercept the old request

- Make a new request
- Receive the response
- Respond to the client in response to the old request

In this approach, the server hides the redirection information from the client. The following server code performs the internal redirection:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   if (q.url === '/foo') {
4.     let data = ''
5.     h.get('http://www.google.com', (m) => {
6.       m.on('data', (d) => {
7.         data += d
8.       })
9.       m.on('end', () => {
10.        r.end(data)
11.      })
12.    })
13.  }
14. })
15. s.listen(12000)
```

What use case does this technique solve? None. The user/client never comes to know about the redirection happening under the cover. So, it adds a permanent burden to the server to manage the additional stale resource references.

So, we need to make the client aware that the reference is old and needs redirection. The following code depicts this:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   if (q.url === '/foo') {
4.     r.end('this site has moved.')
```

```
5.   }
6. })
7. s.listen(12000)
```

And accessing the server from the browser shows the following message:

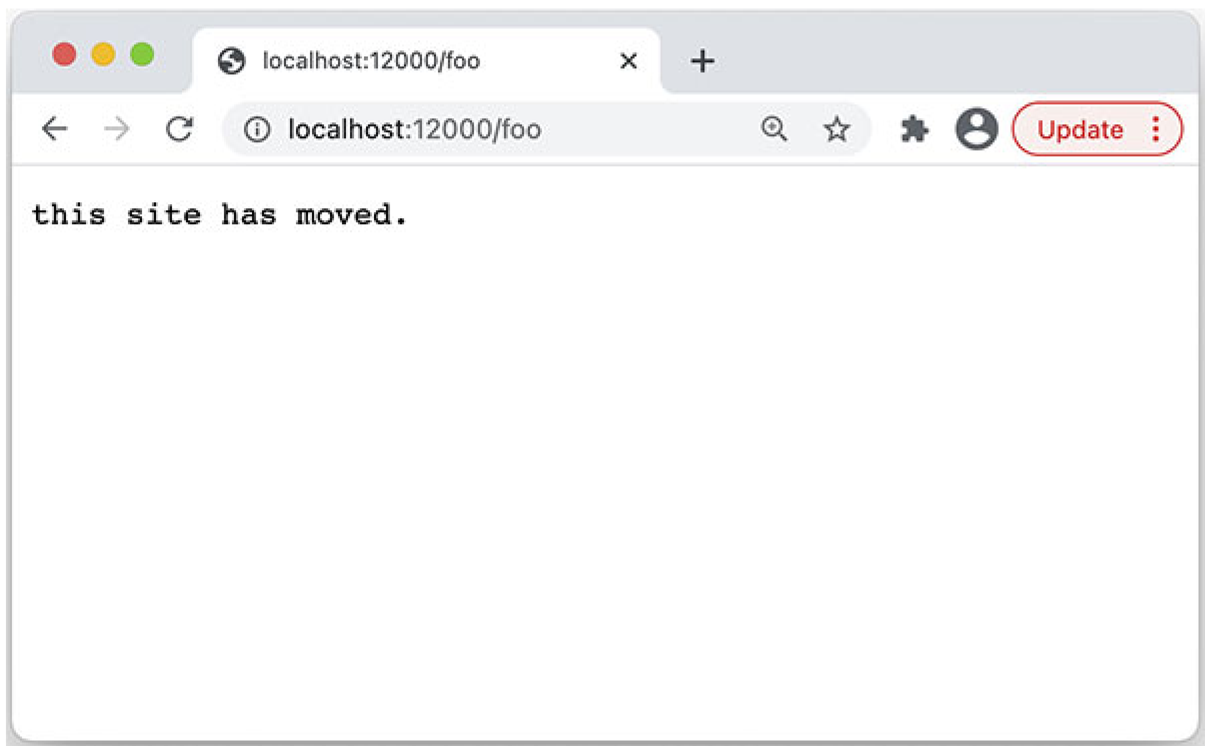

Figure 6.13: *Accessing a moved site route*

This does not go well with clients and leads to poor user experience. Returning the new reference looks like a better solution, but it still isn't the best. The following code sends more useful information to the client:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   if (q.url === '/foo') {
4.     r.end('this site has moved. pls use ' +
5.     'http://www.google.com instead')
6.   }
7. })
8. s.listen(12000)
```

And the following screenshot shows what the client receives:

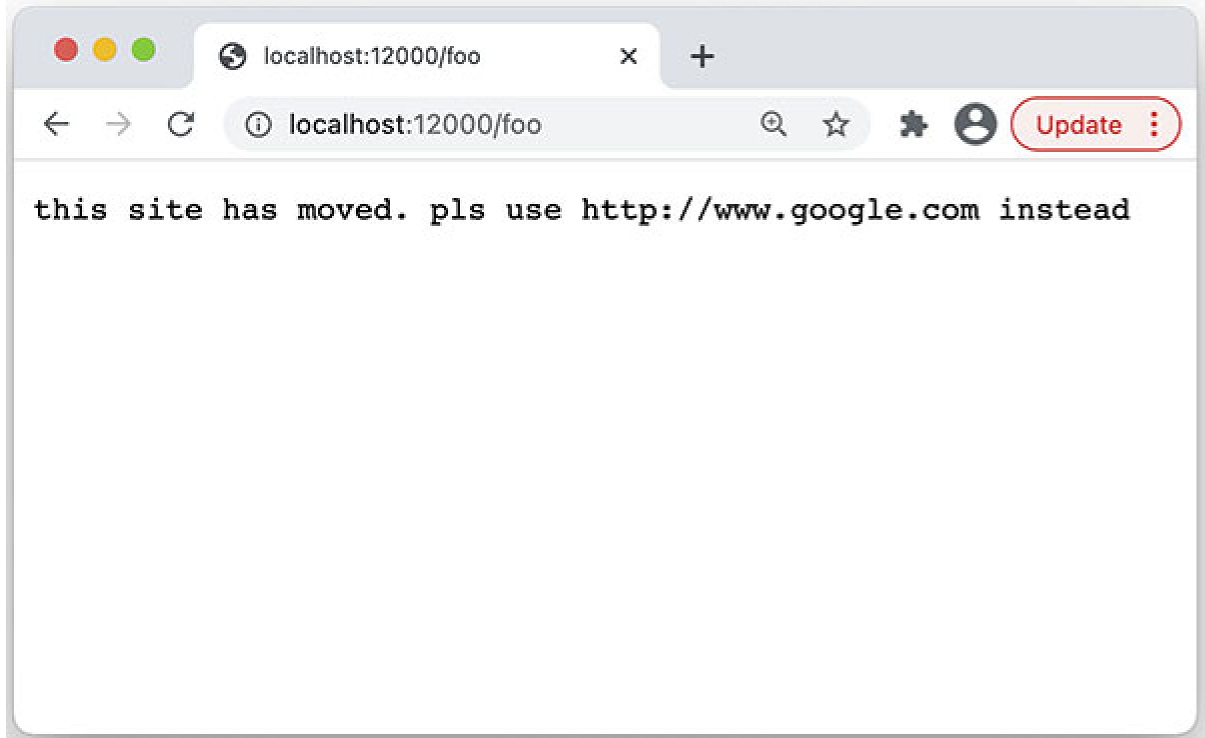

Figure 6.14: *Accessing a site with moved site route*

So, the one that works better is 'co-ordinate with the client and redirect automatically', but let the user know that this is happening so that they can refresh their bookmarks.

The following code illustrates the usage of status code for redirection:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   if (q.url === '/foo') {
4.     r.statusCode = 302;
5.     r.setHeader('Location', 'http://www.google.com')
6.     r.end()
7.   }
8. })
9. s.listen(12000)
```

In this case, a browser client understands that the requested endpoint has moved, and the server wants it to re-request to the new location that is a replacement of the old one. Subsequently, the browser performs the re-request to the new location, transparent to the user. The user sees the redirection message momentarily, and from the new target address that appears in the address bar, the user becomes knowledgeable about the movement that they can accommodate in future uses.

The seamless redirection performed by the browser client is demonstrated with the help of a JavaScript console window, wherein the HTTP headers pertinent to the request and response are captured.

The following screenshot of the client request details from a JavaScript console shows the header values:

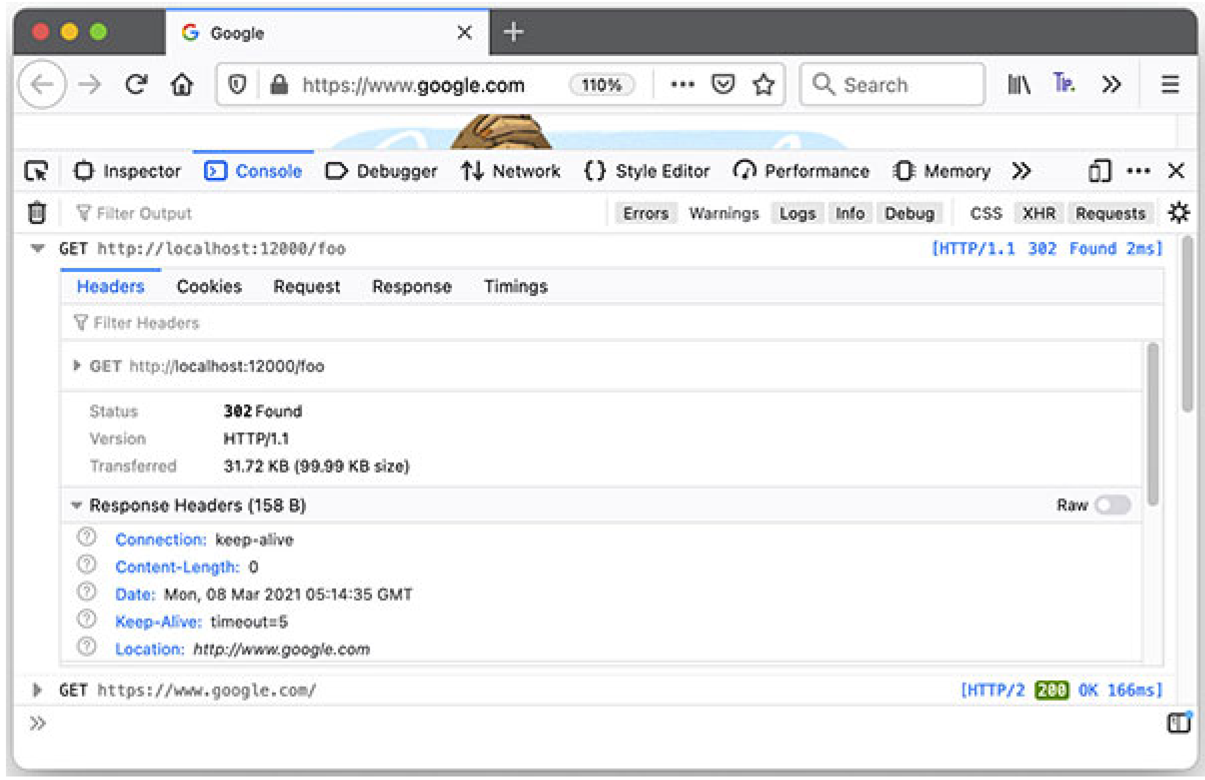

Figure 6.15: *A browser client that handles server redirection*

Multipart form-data

So far, we have been discussing the server serving content to the client. The reverse is a valid scenario as well, for example, file upload. How does the client make the server learn various attributes of the file content being sent, given that the physical file is newly recreated in the server? Multi-part form-data is defined to cover this aspect.

Use case

A website application receives files from clients that need to be stored at server locations. When the files are stored, the server wants to make sure that not only the file content, but also the file metadata are preserved in the server. Some attributes might have changed (for example file creation/access time), but static attributes like filename, extension, and so on need to be preserved.

Definition

The multipart/form-data is a content-type in a client request that represents the submission of form data. This can be used by clients as a way of returning a set of values from filling out a form in a page. A multipart/form-data body contains a series of parts separated by a boundary, that is a series of special characters like CRLF, "--", and a "boundary" string, any sequence of characters conveniently chosen.

The following diagram shows the client-server interaction with a complex user form/file being uploaded to the server:

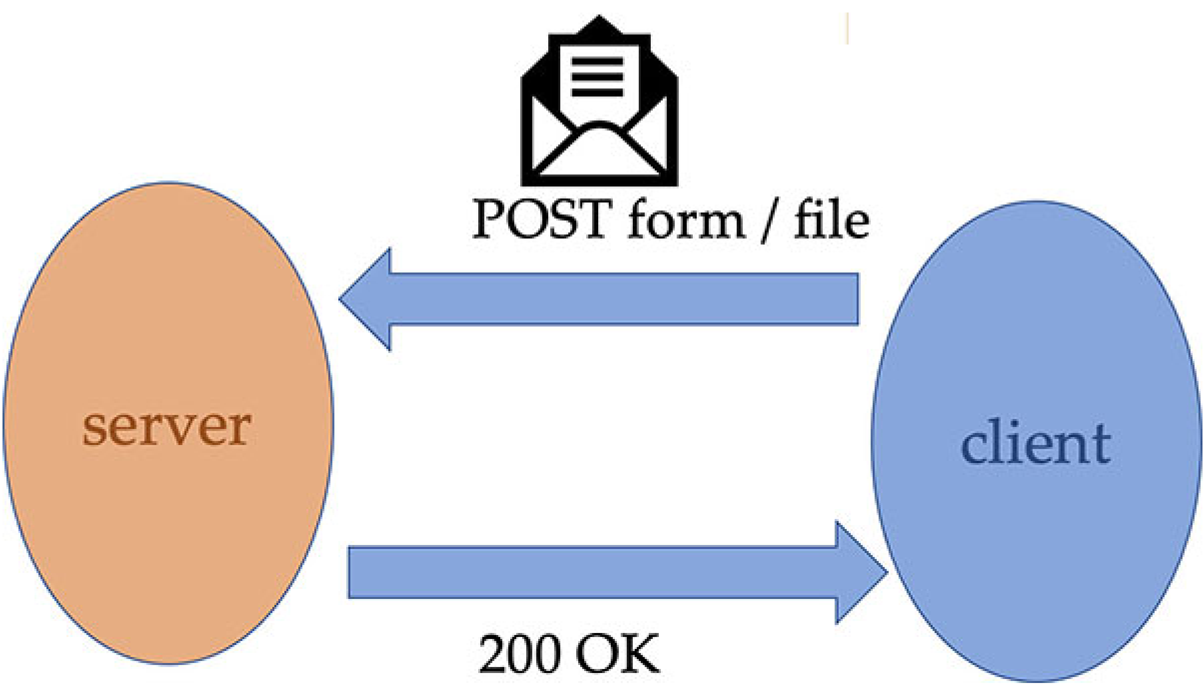

Figure 6.16: *A client-server interaction that uses form data*

Implementation

First, design an approach to how the client would send the file data to the server. Then, learn how the data will arrive at the server. After that, develop a parser to parse the relatively complex header for multi-part form-data. Extract the file content and file metadata from the request. Decide where to store the obtained file, create the file with the same attributes, and store the data in the file.

The following code illustrates basic handling of form/file uploads:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   if (q.url === '/upload') {
4.     q.on('data', (d) => {
5.       console.log(d.toString())
6.     })
7.     q.on('end', () => console.log('done uploading'))
8.   } else {
```

```
9.      const form = '<form action="/upload" ' +
10.     'enctype="multipart/form-data" method="post"> '+
11.     '<input type="file" name="upload"> ' +
12.     '<input type="submit" value="submit"> '+
13.     '</form>'
14.     r.end(form)
15.   }
16. })
17. s.listen(12000)
```

The print in the server shows the following form data that was received from the client:

```
#node multipart.js
-----------------------------144292121922003105253691352O
Content-Disposition: form-data; name="upload"; filename="hello.txt"
Content-Type: text/plain

hello world

-----------------------------144292121922003105253691352O--

done uploading
```

Figure 6.17: *A form data with the associated HTTP header*

The obvious question is, how do we meaningfully extract the content from the form data? There is a lot of protocol-related content, and the actual data is fully blended with the metadata. It will be a pain for a server to manage this complexity while handling every request it receives. How do we better handle complex form data that arrives from the client?

Body parser

Parsing the request body is a non-trivial task, and it needs to be performed in-line with the request before the request is presented to the backend application. This section illustrates specific use cases and the implementation of request body parsing.

Use case

The web server wants to handle complex client requests such as multi-part form data (as explained earlier). The basic Node.js parser does not parse such requests completely, and the request body is made available as is in the request object. Additionally, pluralities of content types are defined in the HTTP specification for data transfer. This means it is desirable to have a higher-level parser that understands common request types and parse those and populate the request object, augmenting the Node.js core parser APIs.

Definition

A body parser consumes the body of a stream-based client request, casts it to an appropriate data type, and attaches it to the request object that the request handler function receives. This saves the request handlers from performing repeated body parsing and helps them focus on their business logic instead.

Here's a diagram that shows the various components a body parser would parse:

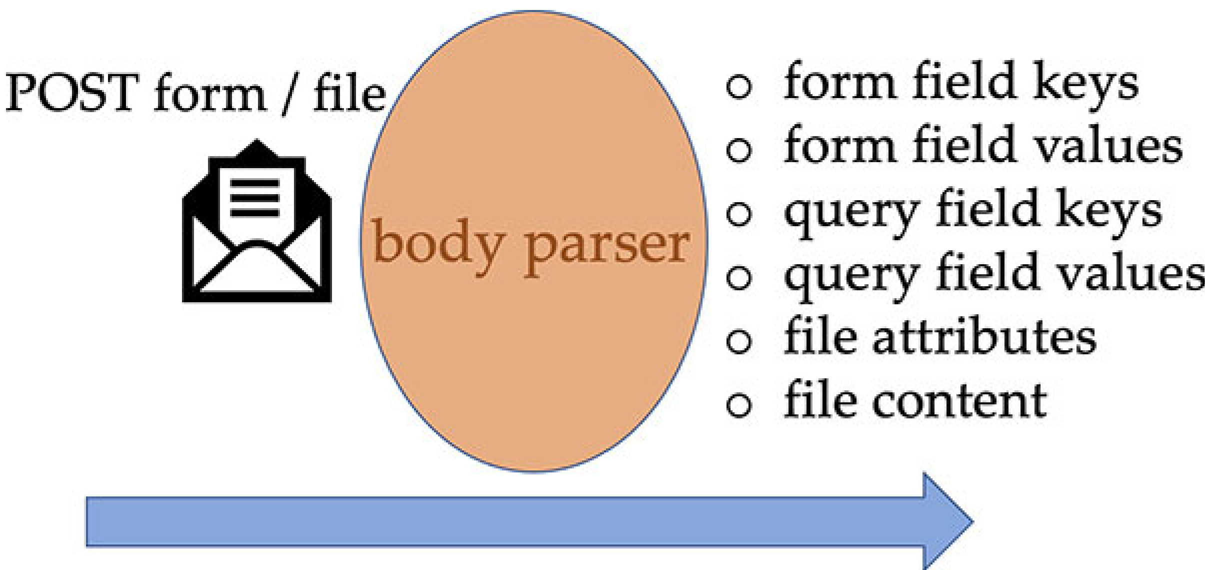

Figure 6.18: *A body parser component with the associated functions*

Implementation

A trivial implementation for the body parser could be an asynchronous function that receives the request object and institutes 'data' and 'end' handlers for it (request being a **ReadableStream** object). In the data handler, the incoming data is either buffered (if the chunk is not enough to parse) or parsed and converted to a known type. Then it is attached to the request body as and when a coherent, fully-formed body is available. In the end, the handler returns to the main business logic of the request handler that wants to consume the body.

A simple body parser logic is as follows:

```
1.  if (q.url === '/upload') {
2.    let data = ''
3.    q.on('data', (d) => {
4.      data += d
5.    })
6.    q.on('end', () => {
7.      data = data.toString()
8.      const delim = data.split("\r")[0]
9.      let file = data.split('filename')[1]
10.     file = file.split('\r')[0]
11.     file = file.replace(/\"/g, '')
12.     file = file.replace(/=/g, '')
13.     console.log(file)
14.     let rest = data.split('\r\n\r\n')[1]
15.     let content = rest.split('\r\n')[0].trim()
16.     console.log(content)
17.   })
18. }
```

The drawback here is that every request handler can be seen as `polluted` with the use of body parser invocations in them. An alternative is to design an abstraction between the client connection callback and the actual request handler callback and let the request pass through the abstraction that parses

the body. When the request handler is invoked, the request object is already populated with the body that is duly parsed.

A reasonable thing to do is to implement a body parser function that is generic enough and can be reused for multiple scenarios. For example, a function that intakes the request and response object, gathers the form data, parses it, and either populates the request object with the parsed content or makes a callback with the parsed content so that the caller can proceed with request handling.

The following code illustrates this improvement:

```
1. const h = require('http')
2. function parse(q, r, cb) {
3.   let data = ''
4.   q.on('data', (d) => {
5.     data += d
6.   })
7.   q.on('end', () => {
8.     data = data.toString()
9.     const delim = data.split("\r")[0]
10.    let file = data.split('filename')[1]
11.    file = file.split('\r')[0]
12.    file = file.replace(/\"/g, '')
13.    file = file.replace(/=/g, '')
14.    let rest = data.split('\r\n\r\n')[1]
15.    let content = rest.split('\r\n')[0].trim()
16.    cb(file, content)
17.  })
18. }
```

Once the reusable parse function is in place, we can use it in our client request handler, as follows:

```
1. const s = h.createServer((q, r) => {
2.   if (q.url === '/upload') {
```

```
3.      parse(q, r, (name, data) => {
4.        r.end(`got ${name} with content ${data}`)
5.      })
6.    } else {
7.      const form = '<form action="/upload" ' +
8.      'enctype="multipart/form-data" method="post"> '+
9.      '<input type="file" name="upload"> ' +
10.     '<input type="submit" value="submit"> '+
11.     '</form>'
12.     r.end(form)
13.   }
14. })
15. s.listen(12000)
```

Cross-Origin Resource Sharing (CORS)

How do we reuse the capability of a second server, under certain circumstances, and yet function safely? Cross-origin resource sharing is a feature that helps in this situation.

Use case

The website wants to "*embed*" data that originated from a second server into its web page, which is otherwise populated by its own web server. At the end of this operation, the resulting web page will appear as a composite page, partly from the main server and partly from the second server. With such resource sharing capability, the website can reuse many such data elements from third-party web services while focusing on its own specialization.

Definition

Cross-Origin Resource Sharing (CORS) is a specification for communicating with a server in a different domain as part of a request with the server in the original domain. The main domain that serves the web

content is called **origin server**, and the second domain that serves part of the content is called **cross-domain**. The specification defines a new HTTP header set. This semantics allows programs to work with the same idioms as same-domain requests while sharing resources cross-origin. The use-case for CORS is simple.

The following diagram illustrates request life cycle with CORS:

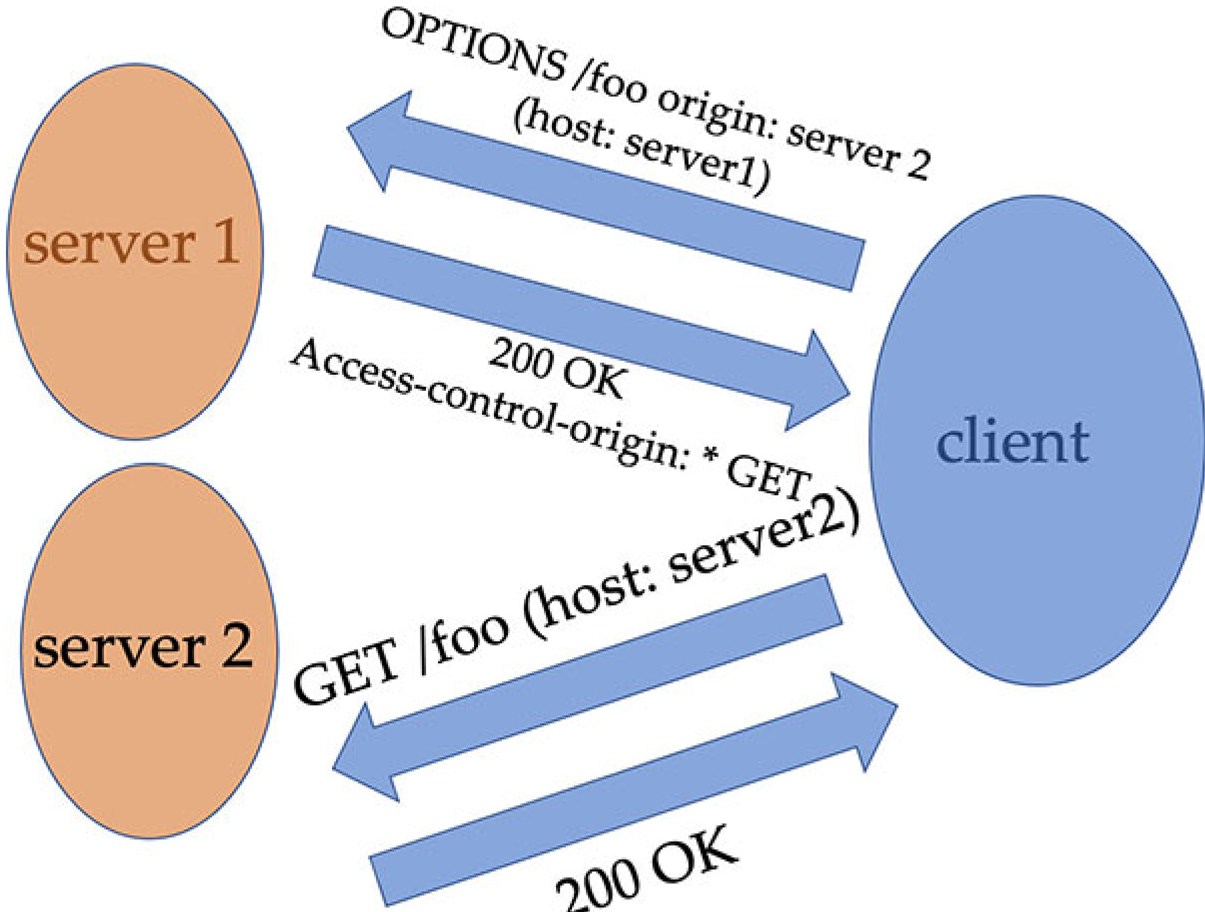

Figure 6.19: *A client-server interaction that uses CORS*

Implementation

First, we need a mechanism for the server to describe what domains are allowed to refer to it. This is achieved by sending HTTP OPTIONS verb to the cross-referenced server.

The following code shows a client that obtains the server's **`OPTIONS`**:

```
1. const h = require('http')
```

```
2. h.request({method: 'OPTIONS', host: 'www.google.com'}, (m)
   => {
3.   console.log(m.headers)
4. }).end()
```

And the following screenshot shows the options available at the server:

```
code — -bash — 49×12 — ⌘1
#node cors
{
  allow: 'GET, HEAD',
  date: 'Mon, 08 Mar 2021 16:27:33 GMT',
  'content-type': 'text/html; charset=UTF-8',
  server: 'gws',
  'content-length': '1592',
  'x-xss-protection': '0',
  'x-frame-options': 'SAMEORIGIN',
  connection: 'close'
}
#
```

Figure 6.20: *Output of the obtained server capabilities*

The client receives the response and parses it. If the response indicates that the origin server has rights to refer the cross-referenced server, the client sends the actual request to the origin server.

Dynamic web page

As is the case with most modern web applications, most of the content is generated at runtime. This section discusses how to implement simple dynamic pages and embed the dynamic part of data into static pages.

Use case

The website wants to generate a page at runtime by parametrizing its content and deriving the values from the current request context. In the most trivial example, after a user performs a login, the result page should

have a reference to the current user. For a second user, this page would refer to the second user, and so on.

Definition

A dynamic web page is a web page that is assembled on the fly. The assembling process is controlled by either the server or the client and may include replacing a placeholder variable with a value obtained in the request context (as explained in the use case) or designing the entire structure of the page with dynamically obtained design elements, or any other customizations in between.

The following diagram illustrates a basic dynamic web page generator:

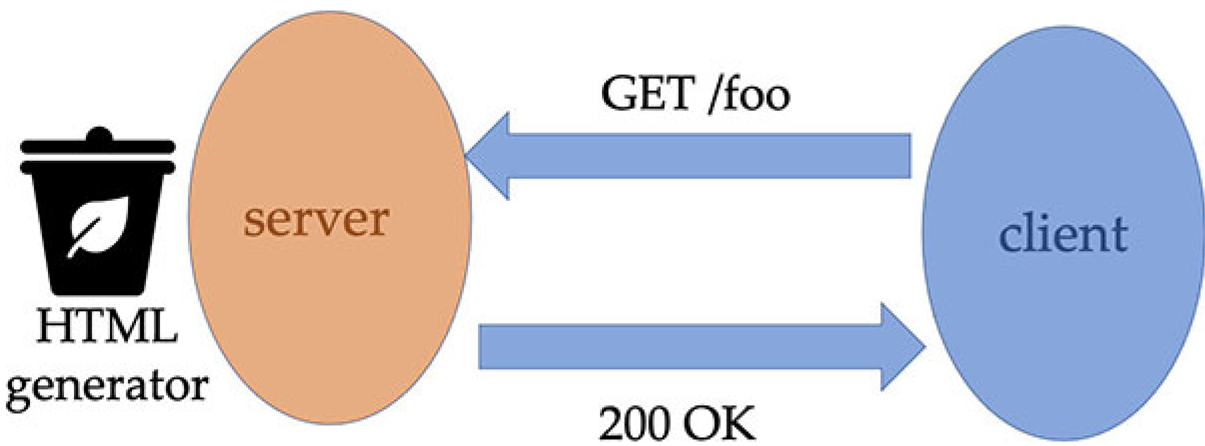

Figure 6.21: *A client-server interaction that uses dynamic HTML*

Implementation

The following code gives the simplest example of a dynamic web page:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   r.end('<html><body>hello world!</html>')
4. })
5. s.listen(12000)
```

This is how it will be accessed through a browser client:

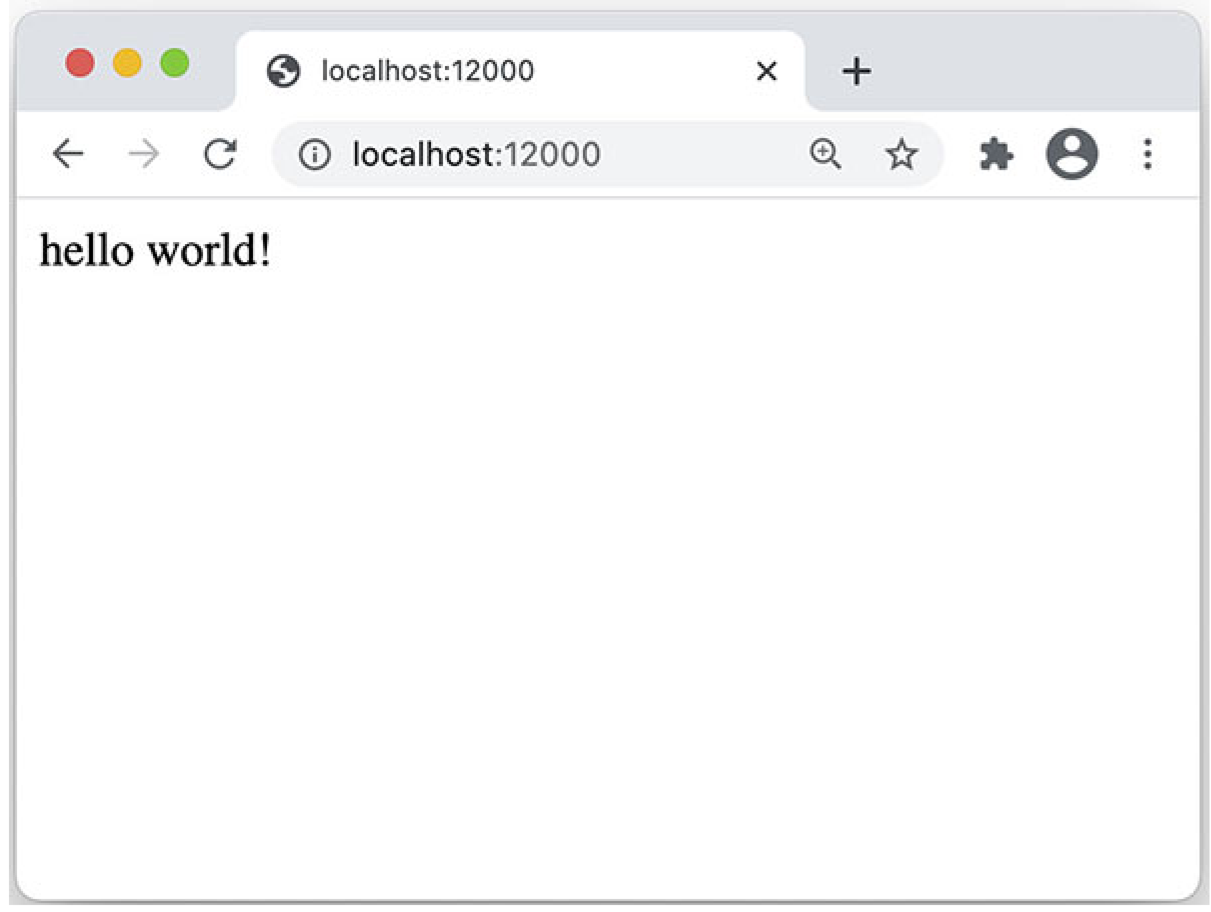

Figure 6.22: *Accessing a server that generates simple HTML*

How do we parameterize it? For example, a logged in username? The following code parses the URL, extracts the username token from it, and prints it back:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   r.end(`<html><body>hello ${q.url.split('=')[1]}!</html>`)
4. })
5. s.listen(12000)
```

And the browser client requests with a URL that contains the query parameters, as shown here:

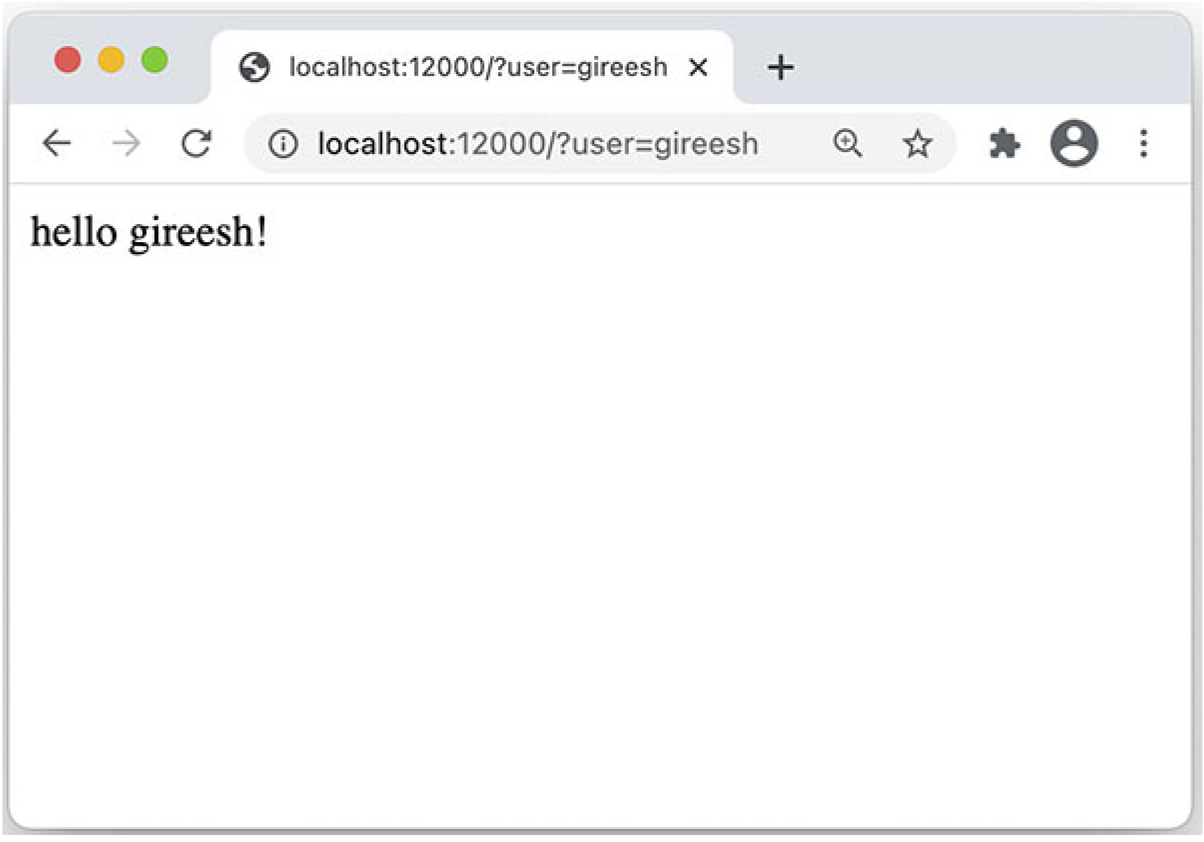

Figure 6.23: *Accessing a server that embeds parameters in HTML page*

How about getting arbitrary records from a database? The following code illustrates random data generated in the server being passed to the client:

```
1. const h = require('http')
2. const m = new Map()
3. for (var i=0; i< 10; i++)
4.   m.set(i, Math.round(Math.random() * 100))
5. const s = h.createServer((q, r) => {
6. let data = ''
7. for(var i = 0; i < 10; i++)
8.   data += `<tr><td> ${i} </td><td> ${m.get(i)} </td></tr>`
9. const html = '<html><body>' +
10.              '<table><tr>' +
11.              '<th> key </th>' +
12.              '<th> value</th></tr>' + data +
```

```
13.                   '</table></html>'
14.    r.end(html)
15. })
16. s.listen(12000)
```

And the client renders the data as a table, as dictated by the generated HTML:

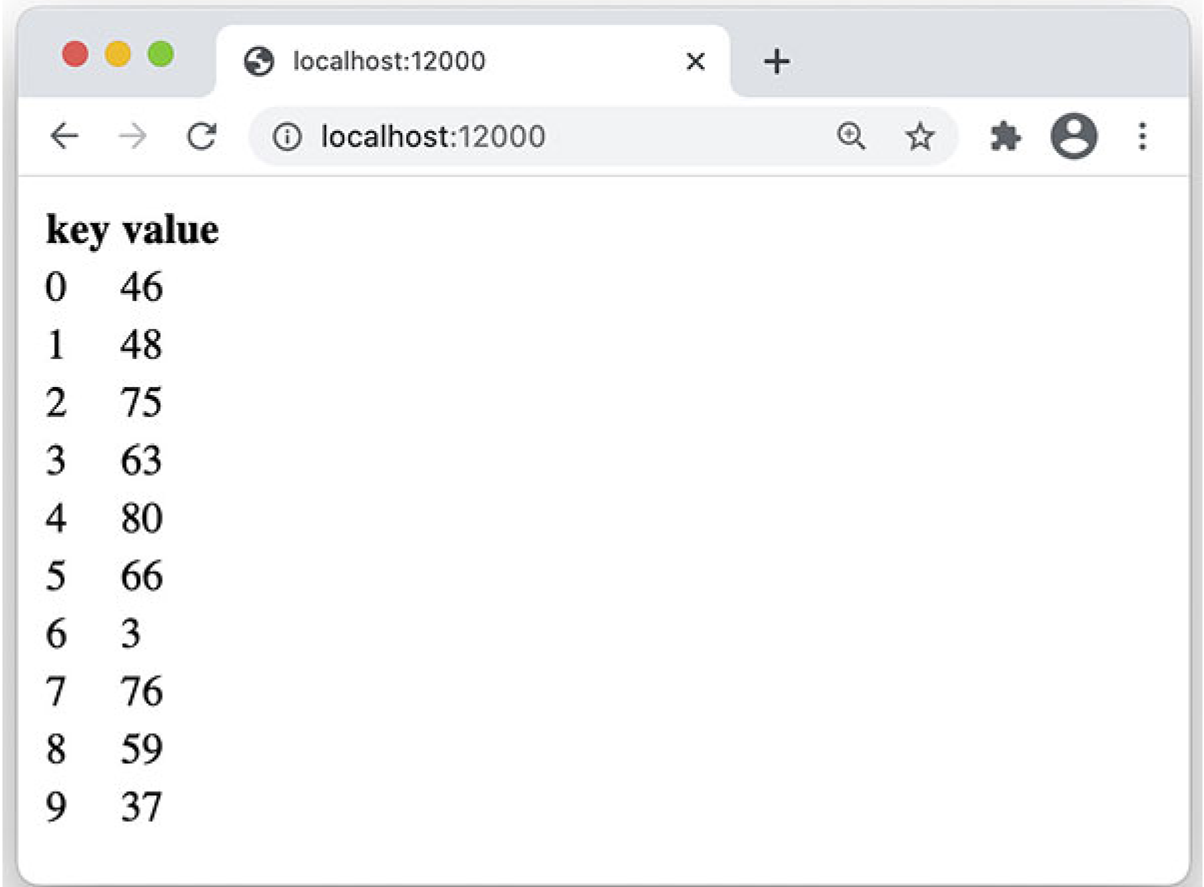

Figure 6.24: *Accessing a server that generates HTML pages with tabular data*

HTTP status codes

Every response from the server should have a status code to indicate the interpretation of the server about how it handled the request. This helps identify the request result outside of the response data and its nature. The status codes are part of the HTTP specification.

Use case

Remember the use case for HTTP request methods? A server can easily classify the type of request based on the request method (verb) and then parse, validate, and process the rest of the request accordingly. In short, an HTTP request method represents the highest abstraction level for the request. When the server responds, it makes sense to have a similar mechanism at the client side—have an abstract verb that represents the type of the response at the highest level, and the header or/and body carry the full details.

Definition

The status code is a three-digit integer code representing the result of a previous request, along with an optional full response. The status code provides a method for HTTP clients to interpret the response reliably and consistently.

Implementation

The response codes are classified into five based on hundreds of possible outcomes of common requests, and a client, which is not interested in covering the full range of scenarios, can inspect the first digit of the response and get the top-level classification of the response.

The following table shows the classification of HTTP status codes:

Code range	Category	Meaning
100 – 200	Informational	Request received, still in progress (further server action)
200 – 300	Successful	Request received, validated, and processed (desirable)
300 – 400	Redirection	The request needs forwarding (further client action)
400 – 500	Client error	The request was not well received
500 – 600	Server error	The request may be good, but unexpected server state

Table 6.2: *Ranges of HTTP status codes*

The following table shows the most familiar status codes:

Status code	Meaning

200	OK
404	Not found
301	Permanent Redirect
500	Internal server error
503	Service unavailable

Table 6.3: *The most commonly used HTTP status codes*

Server security

Server security is one of the most important aspects of the web application. Let's understand common security issues that are relevant to a web application and the recommended remediation to each.

Use case

The server is centrally placed and every client request passes through the network, and the server has no direct way of validating the user's authenticity and credibility, so the server (as well as the client) is inherently subject to security threats at various dimensions. Every component involved in the whole transaction needs to be hardened against these threats.

Definition

Server security is the act of securing the web server, website, and web services from various internet-based security threats. By virtue of the placement of this software on the Internet, security exploitation is a vast topic with wide possibilities. As a result, security best practices that addresses these threats have evolved and grown to become a branch of software engineering itself.

Implementation

Security threats can be classified into three types based on how they impact the server:

- Theft, corruption, destruction, and disclosure of server data
- Illegal access and usage of server resources

- Denial of service

The following are some common security issues that a web server can be subjected to, along with their practical remedies:

Data privacy and integrity

The basic design of the Internet (connected network of computers) allows every connected computer access to every byte of data in the network, irrespective of the computer's role in the data transport. So, if a client makes a request to a server and the server responds with some data, the transaction details as well as the data contained in the transaction reach every system in the network.

Threat

A malicious player can read and understand a transaction and exploit it by faking the client, the server, or both.

The following diagram shows the Internet in the middle of a client-server interaction, along with the associated vulnerability:

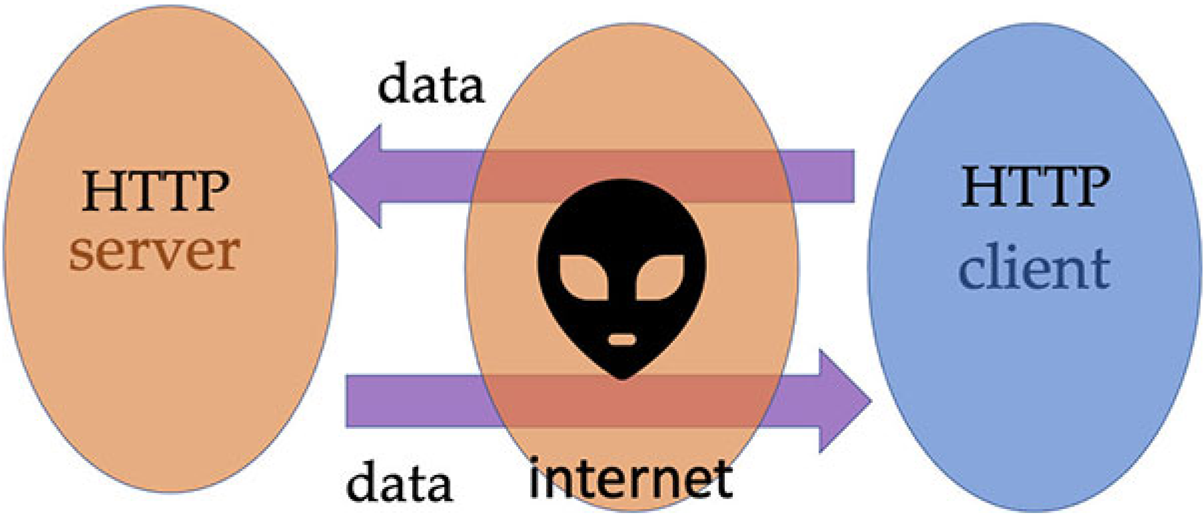

Figure 6.25: *Client-server interaction through the Internet*

Remedy

HTTPS (HTTP Secure) is an extension to HTTP that is used to secure the data used in the HTTP communication. It operates on top of HTTP and creates an abstraction of a secure channel in an insecure network. This

means the data still flows normally, but it will be encrypted using strong cipher algorithms that can be decrypted only by its intended receiver.

The following diagram illustrates the usage of secure protocols to protect the data pertinent to the client-server interactions:

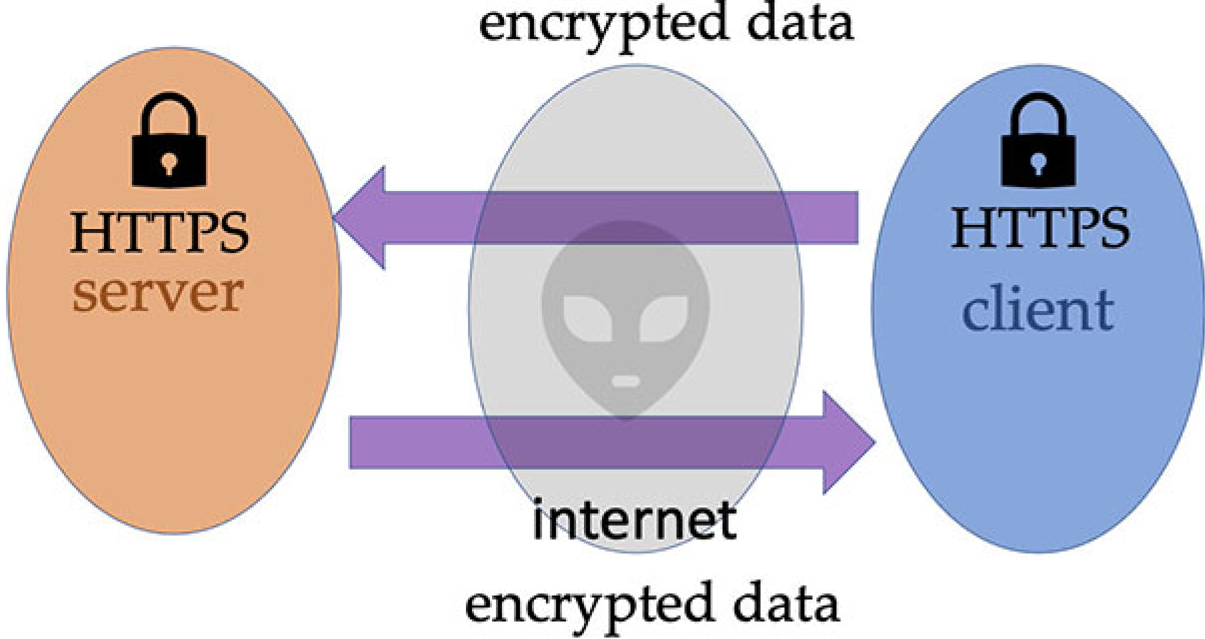

Figure 6.26: *Client-server interaction with encrypted data*

Cross-Site Scripting (XSS)

Assume that the web server has a workflow by which the user request content is also returned to the client with its response. Such a web server is vulnerable to cross-site scripting attacks.

Threat

In the simple form of the attack, a URL is presented to the user through email or some other media. A URL contains crafted messages, such as JavaScript. When the user clicks on the link, the server performs certain actions through the said workflow, but it eventually returns the script as part of its response. This causes the client browser to render the HTML while executing the script as part of the rendition. Now, depending on what is coded in the script, the attacker is able to perform arbitrary insecure operations in the client system.

The following diagram depicts this:

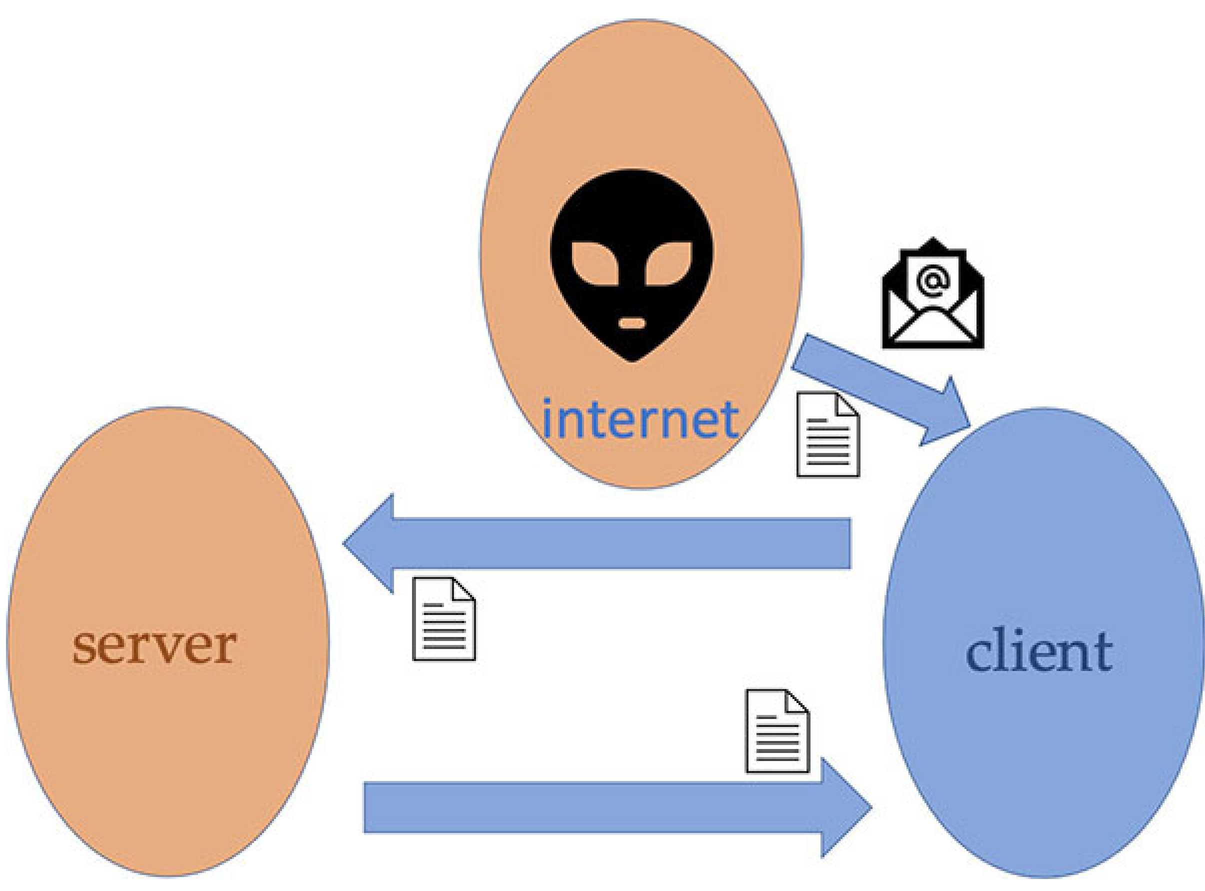

Figure 6.27: *Client-server interaction with cross-site scripting*

Remedy

How can we remediate the vulnerability? The most straightforward way is to:

i. avoid reflecting user content back to the client

ii. if that is absolutely unavoidable, institute a special routine that parses the request and suppress or quarantine any script elements.

The following figure illustrates the measures for preventing cross-site scripting:

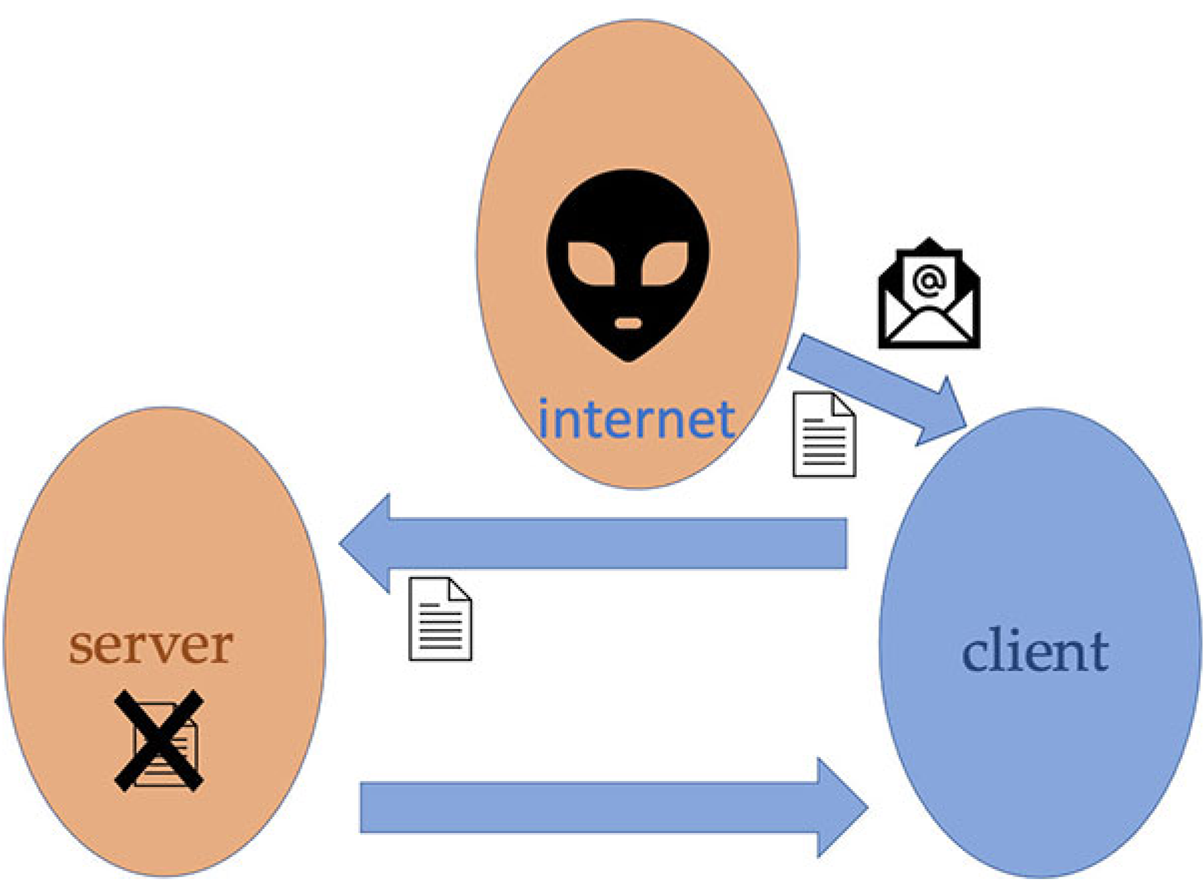

Figure 6.28: A server that validates and normalizes scripts in requests

Query injection

In a data-driven application, with the server making queries to a secure backend database, the query parameters can be derived by user requests. In an over-simplified example, this refers to fetching a user record from a secure database and returning it to the user while the name of the user is used to uniquely identify the record; the username is obtained from the user.

Threat

A malicious user can supply a different username or a wildcard (a regular expression that would still make a meaningful query semantics to the target database when evaluated, expanded, and augmented with the main query string) that matches every user in the database, and thereby obtain sensitive data, even though the entire infrastructure for data transport is secure.

The following figure illustrates query injection vulnerability:

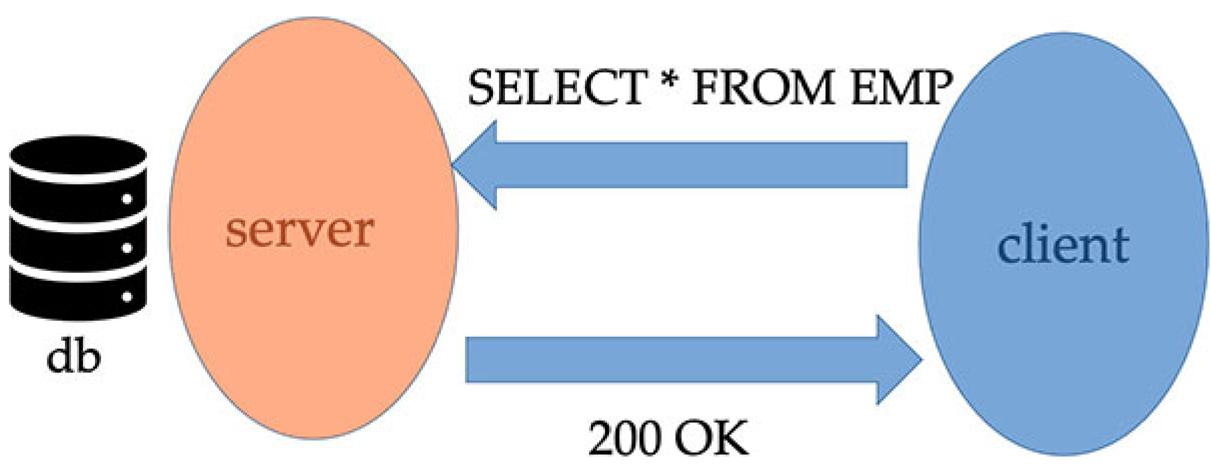

Figure 6.29: *Client-server interaction with arbitrary database query strings*

Remedy

Common remedies include input validation of query string and defining database permissions. For example, define the query string with parameterized placeholders derived from the user input, and ensure that the user input matches the expected type of the parameter rather than unbounded query phrases or regular expressions. From the database side, we can define fine-grained permissions to types of requests to sections of data, which ensures that unbounded or malicious requests will not be honored.

The query validator component, as shown in the following figure, offers protection from such threats:

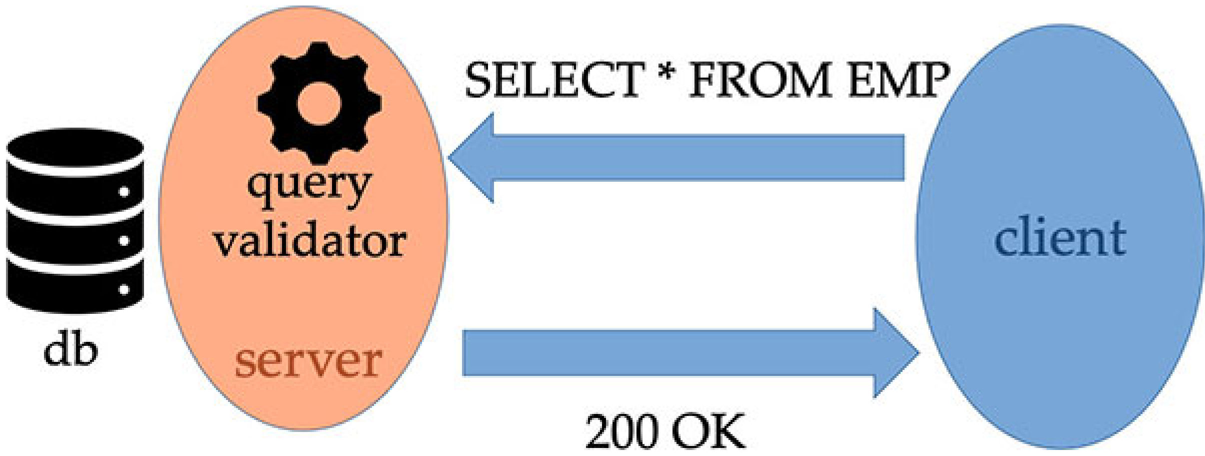

Figure 6.30: *A server that validates and normalizes query strings before execution*

(Distributed or non-distributed) Denial of Service

In this type of vulnerability, the objective is to cause the server software, hardware, or the network to fail to function either temporarily or permanently, leading to denial of the service to the server's users.

Threat

The attack is performed by carrying out a careful study of weak points in the server's execution environment and exploiting that to cause the damage. The attack surface for this threat is wide.

The following figure shows how the attack can cause reliability issues in the server:

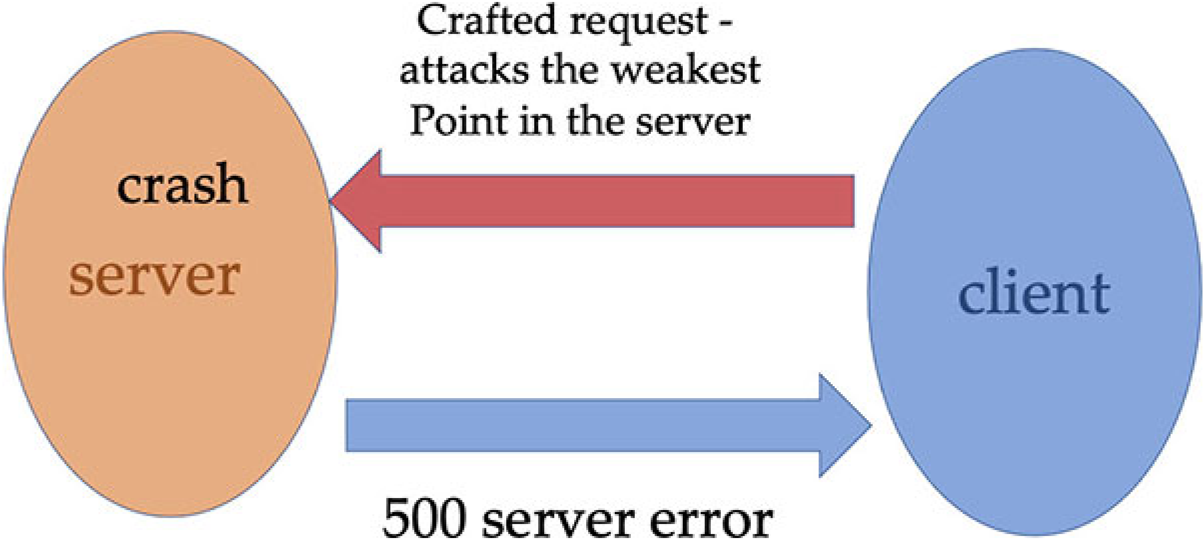

Figure 6.31: *Client-server interaction with denial of service vulnerability*

Remedy

There is no single remedy to this threat. Every part of the execution stack of the server needs to be carefully designed to ensure reliability and durability. Some best practices are:

- **Input validation**: Ensure that combinations of user input can be properly validated and sanitized before processing and the code flow path that process the data is free from abnormal termination due to the data.

- **Isolation**: Ensure that the server stack is isolated from the traffic through a dedicated software for managing the traffic (such as reverse proxy or load balancer).
- **Firewalling, switching, and routing**: Perform some level of sanity check on the requests and ensure that they are pruned before presenting to the server and are not presented in their raw form.

The following figure shows how to remediate from such attacks:

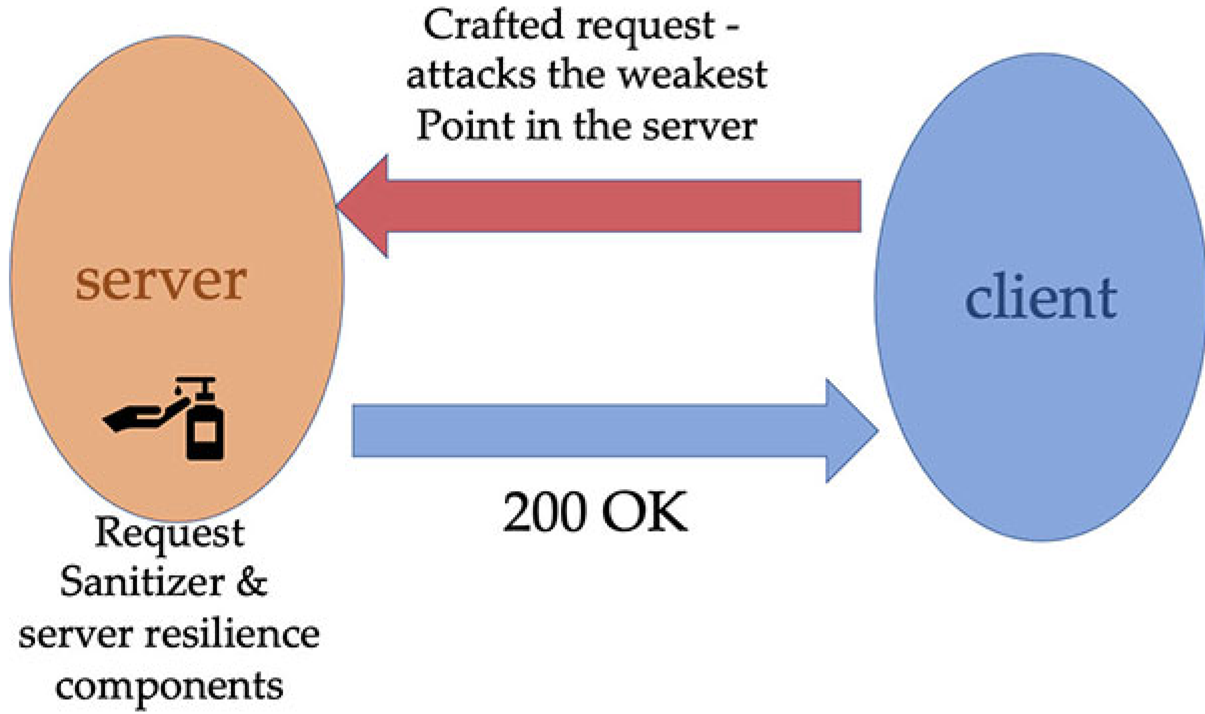

Figure 6.32: *Additional measures in the server to improve reliability*

Brute force (weak authentication)

Brute force is also referred to as repeated, random guesses of the credentials. The objective is to gain access to the system through impersonation.

Threat

In this type of threat, an attacker uses structured, empirical, or/and statistical means to `guess` another user's password and uses it to hijack their account. Despite all security measures, the credentials directly help the

attacker impersonate a genuine user, invalidating all other forms of security controls.

This is illustrated as follows:

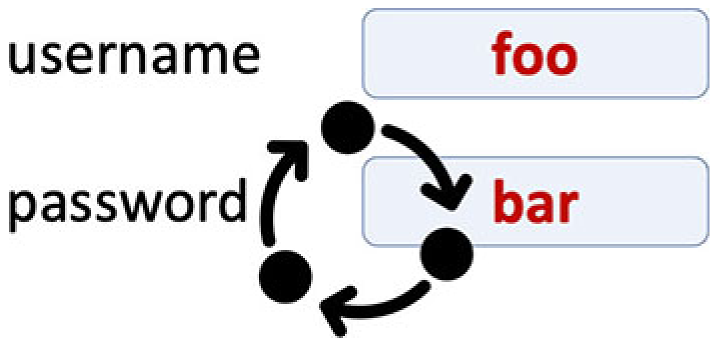

Figure 6.33: *Iterative password guessing method*

Remedy

There are a number of best practices around password strengthening, such as:

i. using a complex password that is hard to 'guess'

ii. managing its storage in a secure manner

iii. using multi-factor authentication (in which password is just one pass, and other passes are also required to actually log in, for which the attacker may not have enough leeway between the first and the second passes etc.)

The following figure illustrates this:

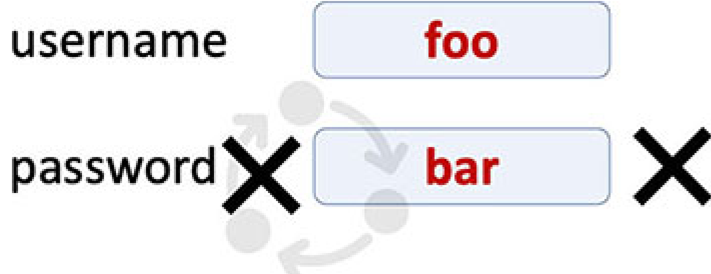

Figure 6.34: *Measures to strengthen both the password and its intake methods*

Conclusion

In this chapter, we looked at the building blocks of a typical web server that deals with the use cases of a website, starting from implementing a static web content server to web server security threat types and their mitigations. This enabled us to develop a website with common features that are functionally correct and commercially feasible and reasonable.

In the next chapter, we will look at the external components that our server typically wants to interact with, such as backend services—either part of our larger application or third-party services we want to leverage and reuse. This will comprehensively cover our web server and website componentry.

CHAPTER 7

Interacting with Backend Components

In this chapter, we will look at other software components that our web server typically interacts with as part of serving our client. These could be services or modules developed as part of our application, third-party services that serve a specific purpose, or components that manifest well-defined design patterns in the software engineering. It is essential to understand where (at what point in the request-response cycle) and how the server interacts with these services. That would make our understanding of the web server complete in all aspects in the backend.

Structure

In this chapter, we will cover the following topics:

- Backend components: **internal services**
- Backend components: **external services**
- Backend components: **database**

Objective

After studying this chapter, you will be able to understand the general objective and design of leveraging backend services in your web application. The discussion will include the architecture of a web application, integration with backend services, and various design considerations of such interactions, like functional, performance, reliability, security, and serviceability. These considerations will help us develop backend components, architect newer interactions, extend the existing interactions, and even optimize service invocations for improving server efficiency.

Backend components: internal services

The internal services of an application are helper routines or modules that are logically part of the application but developed and deployed separately to achieve maximum decoupling, leading to overall efficiency improvements to the application's development and production life cycle.

Intent

The reason for our web application to interact with internal services is largely attributed to microservice architecture. When the web application was a single process, it had faced several issues throughout the lifecycle of the application, namely:

- **Development, testing, and deployment**: Due to the monolithic nature, these phases have to 'drag' all the components in the monolith together, even for a minor change in the application.
- **Readability and maintenance**: Due to its sheer size, possibly tangled with unwanted abstractions in the modules as well as several patches and extensions, reading code can be extremely difficult, and it also brings in the possibility of introducing bugs.
- The application's footprint grows unnecessarily.
- Minor issues in an unimportant part of the code can bring down the entire process.

The micro-service architecture was evolved to remedy this; it decomposes large applications into small, self-contained modules that can function (bootup and execute and undergo process life cycles) independently.

The following diagram illustrates micro-service architecture:

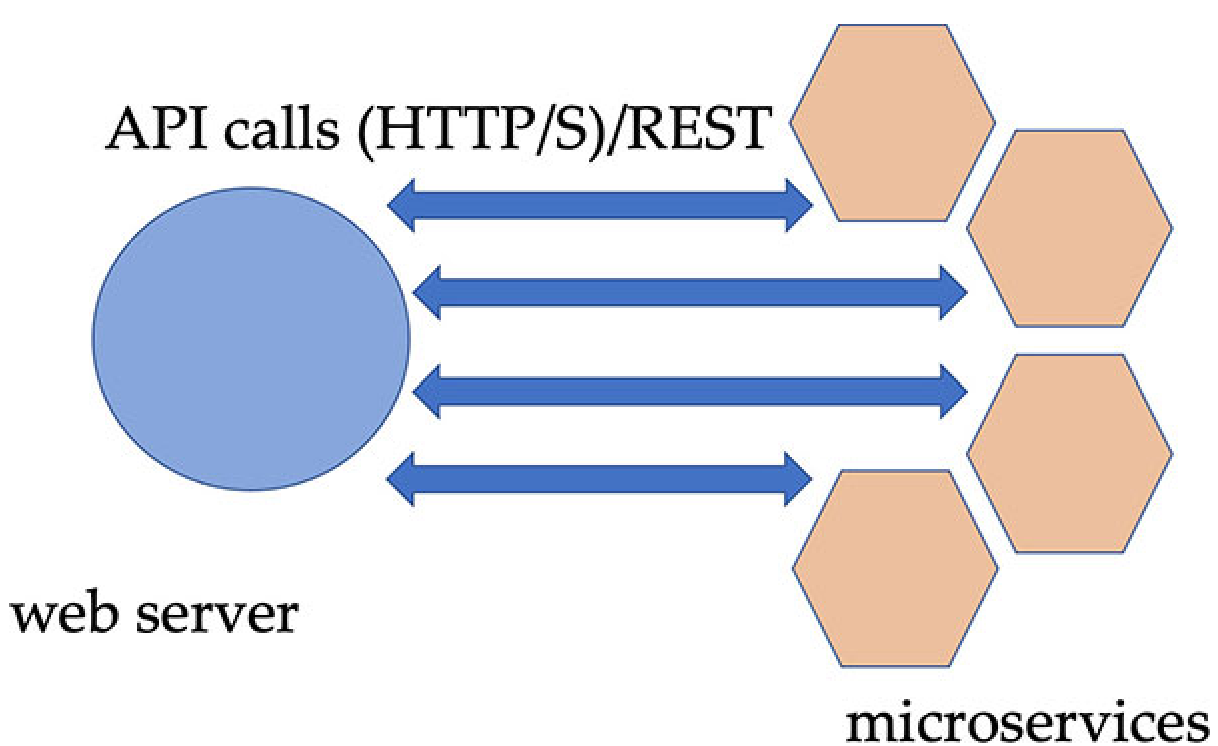

Figure 7.1: *Microservice architecture*

A side effect of this architecture is that a direct function call from a module A to a module B in an earlier monolith is now a service invocation across the network in the microservice world.

The transformation of a monolith into a microservice is depicted in the following diagram:

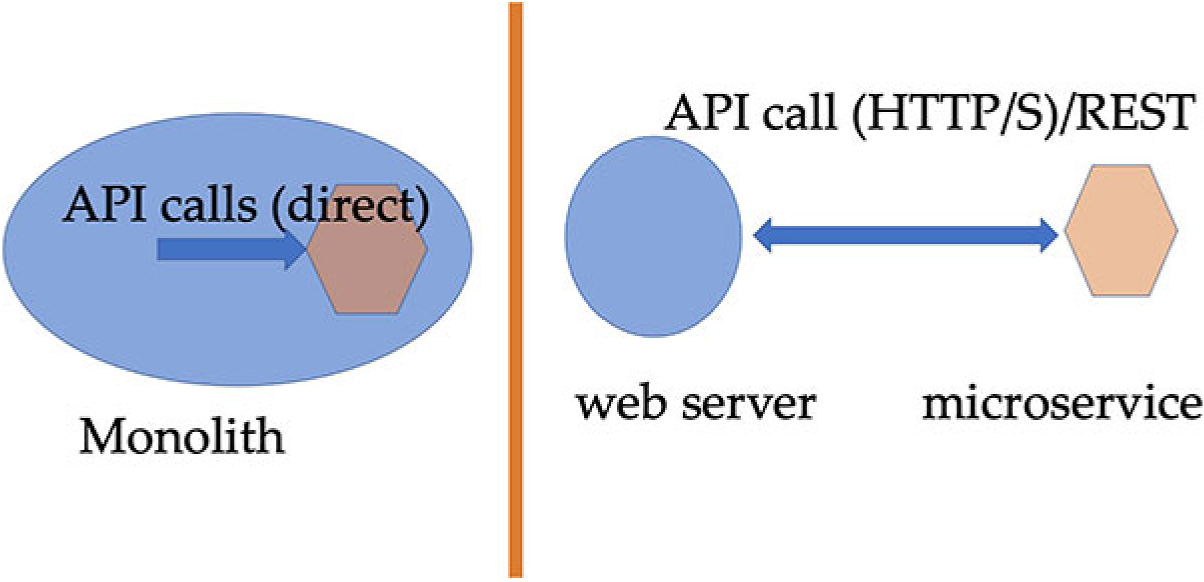

Figure 7.2: *Transformation from monolith to microservices*

Design considerations

In this section, we will look at design considerations for the web application in view of various parameters like capability, performance, scalability, reliability, and so on. We will also establish and ratify the way the internal modules are architected.

Functional

When interacting with an internal service, one of the primary considerations is how the interaction will be carried out. We have already seen that the web server functions with a request-response cycle, which means it receives the client request, parses and processes it, prepares the response, and then sends the response back to the client. With the service invocation in place, its role would most naturally be in the middle—contribute to preparing the response. It is possible that the internal service also needs the request context and details either in parts or in their entirety.

The following figure illustrates the placement of service invocation in the request-response cycle:

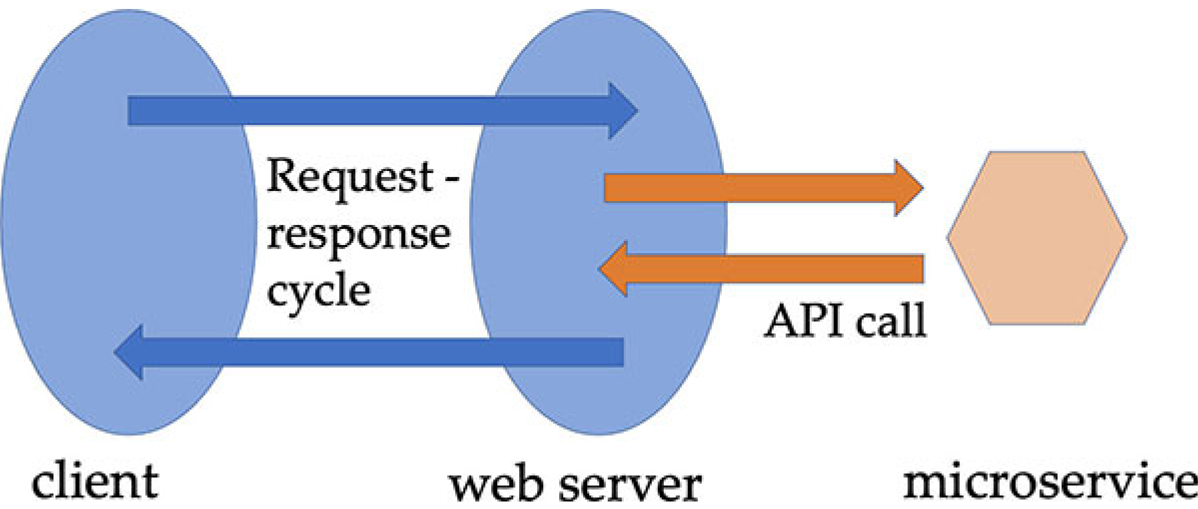

Figure 7.3: *Microservice call melded into a request-response cycle*

With our asynchronous, event-driven programming model, the service invocation scenario can be designed as a chain of callbacks, wherein the client request handler callback initiates the service request, the response handler of the service request installs a response end callback, and the final response is made to the client in the response end callback.

This is illustrated in the following code sample:

```
1. const h = require('http')
2. const f = require('fs')
3. const s = h.createServer((q, r) => {
4.   h.get('http://localhost:13000/myservice', () => {
5.     const data = '';
6.     m.on('data', (d) => {
7.       data += d
8.     })
9.     m.on('end', () => {
10.      r.end(data.toString())
11.    })
12.  })
13. })
14. s.listen(12000)
```

As we can see, the service invocation is fully melded into the web server and is an integral part of the request-response cycle. A positive side effect of this is that the integration with and the invocation of the service becomes seamless with respect to the server's function. On the other hand, this leads to the concern of the server code becoming subject to general software efficiency indicators like performance, reliability, security, and serviceability.

Tip: A service invocation in a web server can be exemplified using a pipeline. An existing pipeline is cut in the middle, and a new pipeline is inserted at the opening. So, the fluid that flows in the pipeline now reaches the first end of the cut, flows through the new pipeline, comes back at the second end of the cut, and continues flowing through the old one till the end.

Performance

An additional set of network transaction between the original client-server interaction causes the performance to be poorer than its monolith counterpart—the same HTTP protocol overheads we discussed in the previous chapters come into play. Additionally, the proximity of the invoking and invoked services can play a role in the network latency, adding to the performance characteristics.

The performance bottlenecks in the service invocations are highlighted as follows:

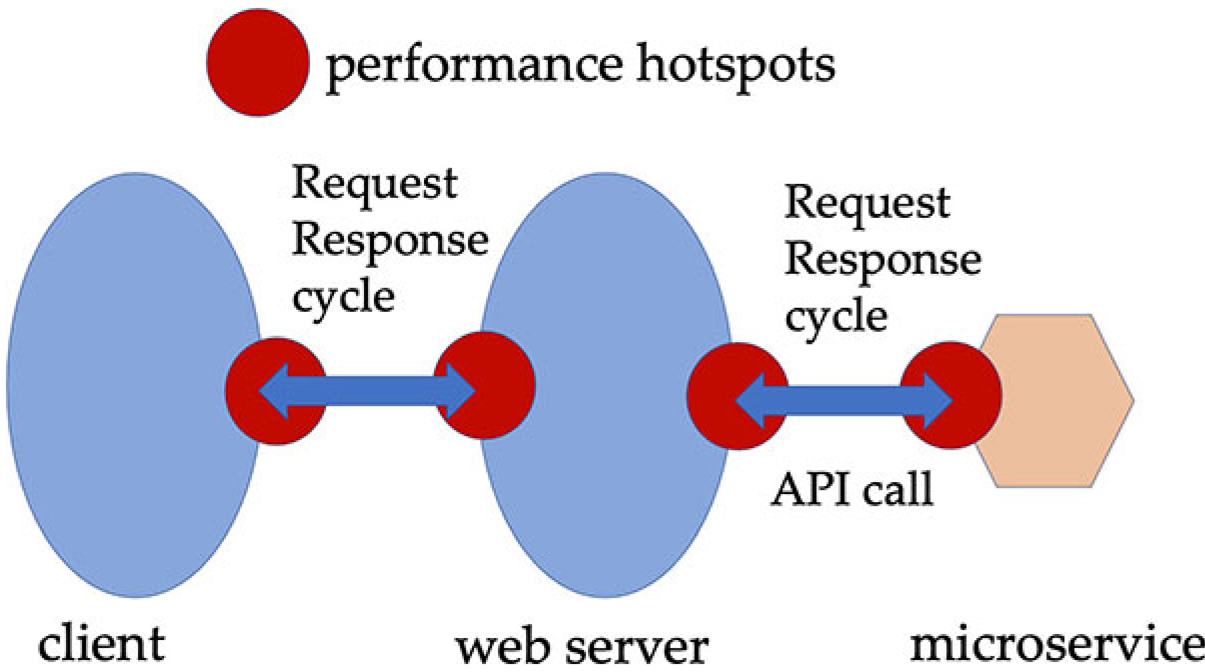

Figure 7.4: *Performance hotspots highlighted in a web server architecture*

Caching of responses is a good remedy to improve performance. Is the internal service insensitive to the request context? Is the internal service independent on data that is modified externally? If so, previous responses can be cached based on the request as the key. This means if some of the request parameters match a predefined set of rules, we can bypass the service invocation and directly compose the response.

The following example code illustrates how caching can help improve performance:

```
1. const h = require('http')
2. const f = require('fs')
3. const cache = new Map()
```

```
4. const s = h.createServer((q, r) => {
5.   if (q.url === 'foo') {
6.     return r.end(cache.get('foo'))
7.   } else {
8.     h.get('http://localhost:13000/myservice', () => {
9.       const data = '';
10.      m.on('data', (d) => {
11.        data += d
12.      })
13.      m.on('end', () => {
14.        r.end(data.toString())
15.      })
16.    })
17.  }
18. })
19. s.listen(12000)
```

Question: The preceding diagram highlights the natural performance inhibitors in a microservice architecture. Assume that your application has reported a severe performance issue. Also, assume that you have identified (through some performance testing) that the time for overall compute is much less than the time for network transport. What strategy will you use for some real performance improvement?

Reliability

What if the target service crashes? What if the target service is not responding? We are in the middle of a request-response cycle when such anomalies occur, and these kinds of events can bring our transaction to an inconsistent state.

The following diagram illustrates this:

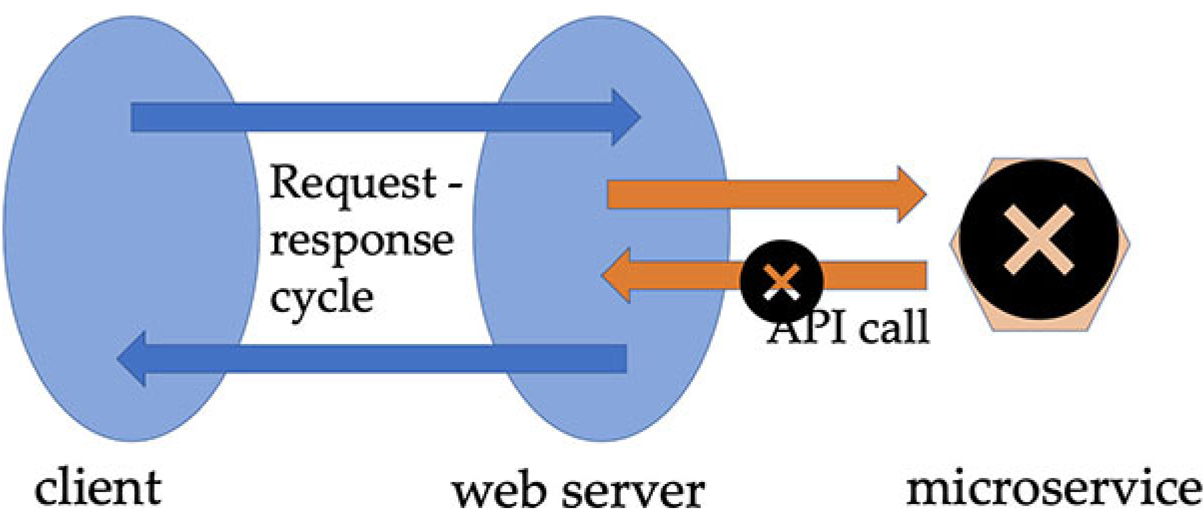

Figure 7.5: *Web server architecture with vulnerable control points*

Defining timeouts, exception catch sinks, and other measures to gracefully manage communications is a best practice. Does the internal service have mechanisms to respond under erroneous conditions with error codes/messages? Does the caller have measures in place to figure out if the internal service failed? If so, the final response can be crafted based on these assertions made at appropriate situations rather than leaving the server in an arbitrary state.

In the following example code, we install an error handler for the invoked service so that we have a well-defined control point if the service fails:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   h.get('http://localhost:13000/myservice', () => {
4.     const data = '';
5.     m.on('data', (d) => {
6.       data += d
7.     })
8.     m.on('end', () => {
9.       r.end(data.toString())
10.    })
11.    m.on('error', (e) => {
12.      r.end(`internal server error: ${e}`)
```

```
13.         })
14.     })
15. })
16. s.listen(12000)
```

Security

A new request-response cycle has emerged from within the original request-response cycle that involves the real client, and the internal service is located in the network as opposed to an in-process module (in case of a monolith), so the communication now becomes a subject of security.

The following figure illustrates additional security measures required for service calls:

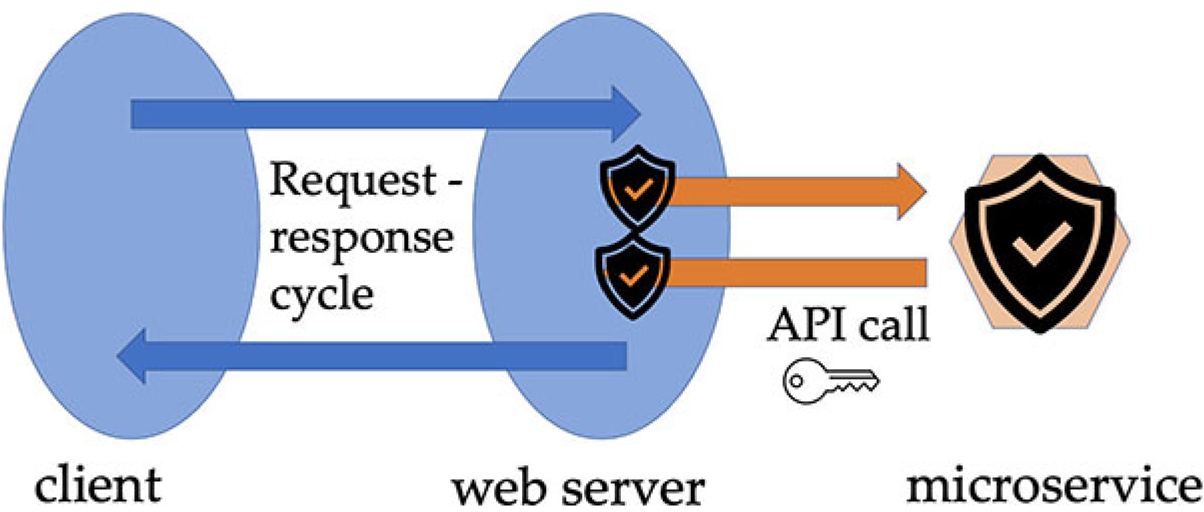

Figure 7.6: *Security hotspots in a service invocation setting*

Applying the same set of security measures that we adopted for a web server component is the best way to address this. Is the internal service located within the same network? Is the system that hosts the internal service managed by an orchestration system? These are some of the considerations that apply when designing secure communication between the server component and the internal service.

Serviceability

An in-process module is now a top-level service running in a separate process in a separate system across the network, with its own process life

cycle definitions, resource consumptions, logging, and such, so the serviceability parameters also need to be redefined. For example, how do we detect and debug a service failure? Which logs needs to be collected and analyzed? Should logs from the consuming and consumed entities be analyzed in unison? Should the logs from both modules be time-series'd so that they can be aggregated or sequenced?

Answers to these questions will take us to observability technology, which is implemented by a number of tools and frameworks. Most of these have dedicated features for applications that follow microservice architecture.

Microservices with observability features are illustrated as follows:

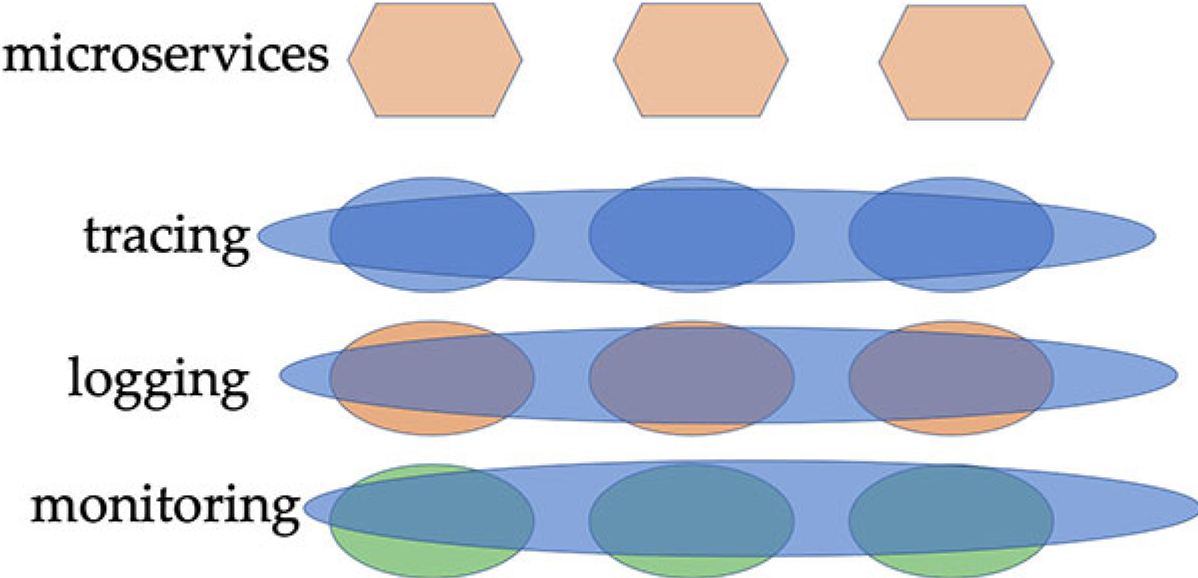

Figure 7.7: *Observability parameters in a microservice architecture*

Question : We talked about refactoring monolith into microservices. Does this mean each service should be developed in the same language as that of the original monolith or the invoking web server? For example, in our scenario of the web server being developed in Node.js, can we have a service developed in Java or C++ and still the communication carried out seamlessly, without additional language-specific bridges or foreign function interfaces?

Backend components: external services

External services are software modules specialized on a specific use case and designed to be highly reusable. In many cases, it makes sense to invoke an existing service to get a specific functionality rather than implementing it by hand. This section examines such external services that our web application can potentially leverage.

Intent

The reason for our web application to interact with external services is attributed to reusability of software and separation of concerns. For example, if you are developing a ticket reservation system, your main concerns are the business logic involved in reservation, such as the amount and type of tickets, seasonal offers, precedence constraints, booking policies, cancellation policies, and so on. But as we know, a vital capability of any ticketing software is the ability to manage payments. Since payment is a pervasive use case in business and not a unique feature of a ticketing system, payment gateway software evolved as a reusable service. They have a well-defined set of life cycle events, transactional constraints, security measures, and policies that can be orthogonal to that of our server.

In summary, it does not make sense to implement a payment feature native to a web server; instead, leverage an existing one that fits its use case and invoke it as a service.

The following figure shows how a web server makes an external service invocation:

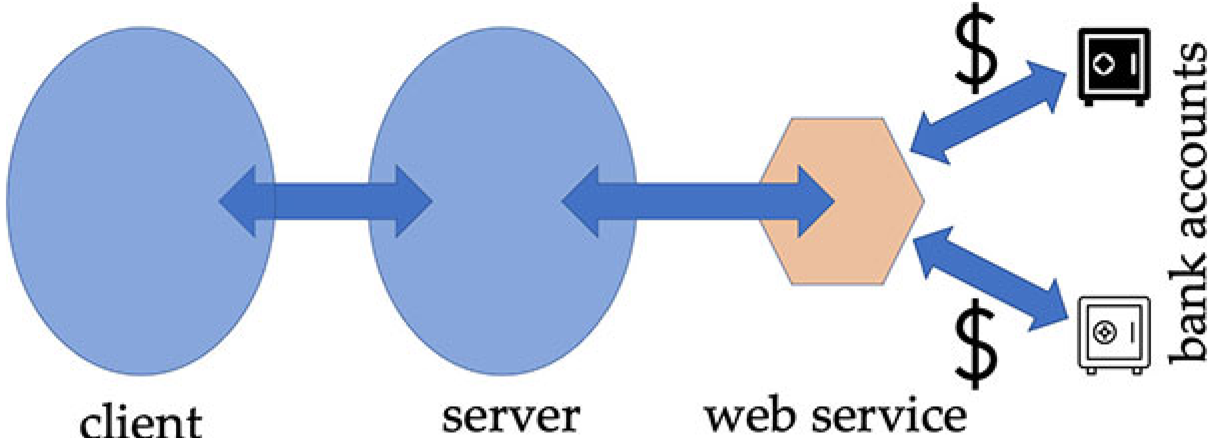

Figure 7.8: *A web server architecture with external backend service*

A side effect of this approach is that the web server application builds a dependency on an external service that is developed, tested, and hosted by a third party, giving rise to a number of implications, such as commercial contracts, **Service Level Agreements** (**SLAs**), parity/disparity with respect to the functional, performance, reliability, security, and serviceability parameters of such interactions.

Here's a typical payment service invocation example:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   const gateway = require('mypayment_gaetway)
4.   gateway.key = '024acd67de99f'
5.   gateway.username = 'user'
6.   gateway.password = 'password'
7.   gateway.card = '1234 5678 9010 1112'
8.   gateway.cvv = '123'
9.   gateway.user = 'foo'
10.  gateway.expiry = '1230'
11.  gateway.comment = 'book purchase'
12.  gateway.amount = 2000
13.  gateway.currency = 'INR'
14.  gateway.toaccount = '987 654 3210'
15.  gateway.uniqueid = 'a4df90bdbbc'
16.  gateway.pay(() => (e, s) => {
17.    if (e != null) {
18.      return r.end(`error making payment: ${e}`)
19.    } else {
20.      r.end(data.toString())
21.    }
22.  })
23. })
24. s.listen(12000)
```

Question: This question can appear anywhere in this book, but we're asking it here for re-iteration, recollection, and better understanding. What is the state of our web server when the payment service is in action? In other words, will the server block all its operations and wait for the payment to be carried out by the third-party service?

Design considerations

In this section, we will look at the design considerations of our application when interacting with an external service. The section discusses implications to functional, reliability, and performance characteristics and illustrates best practices.

Functional

When interacting with an external service, the key consideration is how the interaction will be carried out—exactly like we discussed in the case of an internal service—by embedding service calls between the request-response cycle and potentially passing the request context to the call after properly sanitizing the data. In addition, the external service may have a calling semantics that is not native to our application, in which case the caller will need to have an 'adaptor' or a 'connector' that manages the communication with the service.

A sample external API invocation is illustrated as follows:

```
1. const h = require('http')
2. const hs = require('https')
3. const u = 'https://api.twitter.com/2/tweets/search/stream'
4. const s = h.createServer((q, r) => {
5.   hs.get(u, {apikey:'a09bfde1'}, (m) => {
6.     let data = '';
7.     m.on('data', (d) => {
8.       data += d
9.     })
10.    m.on('end', () => {
```

```
11.         r.end(data.toString())
12.       })
13.     })
14. })
15. s.listen(12000)
```

A key difference between the invocation of internal and external service can be the protocol: an internal service can be used using plain HTTP protocol, while an external service typically needs much higher-level and specialized protocols, like REST, SOAP, XML, and such.

Another embodiment of an external service is an API key or a token. What is the commercial bearing of your web server with a third-party service? Why would it respond to your request and provide an important service free of cost using a software they have developed with great research and carrying intellectual property? So, the calls need to be accounted and billed. An easy way to achieve this is through API keys. A service's consumer service logs into the vendor's website and uses the service. When they make the payment, they receive an API key, an authentication token that vouches for the authorization to use the API for a predefined number of invocations.

Performance

An additional set of network transaction is involved between the original client-server interaction, so the same performance degradation that we saw in the case of an internal service applies here. In addition, external services are almost always hosted on different servers, so network latency can be more than that of internal services. Further, the caller doesn't have enough control over the performance characteristics of the called service as the former has no control over the latter's concurrency, traffic policies, performance considerations, and such. So, a best practice is to ensure that the service is tested under a wide variety of conditions to see that the responses we get are well within the acceptable latency levels.

Reliability

The same reliability measures that we adopted for internal services apply to external services as well, in addition to the network outage possibilities on the internet as our third-party service sits across the network on the internet.

Another aspect to look at is how we want to manage these services' failures. Do we want to relay the same error messages back to the end user upon a failure? Or do we want to abstract the error and provide a more meaningful, contextual message that the end user is able to interpret more easily?

Security

Given that the communication between the web server and the third-party service is now happening across the network, applying the same set of security measures that we adopted for the web server component is the best way to address it. In essence, we should treat our service program as a client to the third-party service, in all aspects, and embed the security measures on the request side as a typical client would do.

Serviceability

Part of the web server functionality is now supplied by a third-party service, so issues from that service or interaction with the service become a new topic of serviceability. For example, if the server fails, how do we identify if it is due to an anomaly in the web server or a problematic interaction with the backend server?

We need to integrate the serviceability features of both the web server and the remote service and make it (the serviceability feature) seamless to the consumer. That way, we can have a uniform problem determination experience irrespective of the problem source.

Backend components: database

Just like any standard application, a web application deals with a lot of business data, and a database software is essential to store those for proper and efficient functioning of the application.

Intent

The reason for our web application to interact with database services is attributed to many important characteristics of the server, like flexibility, performance, reliability, security, and auditing.

- Isolating data management functions from business logic leads to improved flexibility of our web server.
- We improve the overall performance and security of our server by using specialized database software that optimizes data management and secures the data access through well-defined policies.
- Ability to persist business critical data in case of a server crash and reduction in the CPU usage of the server software means our server is now more reliable.
- It is easy to audit the business over a period if we have the data and the data access points separated and powered with special purpose logging mechanism.

In summary, it is advantageous to use a database component to store the business data pertinent to the server, including the session data we studied in *Chapter 6, Major Web Server Components*.

The following figure shows web server architecture with a database backend:

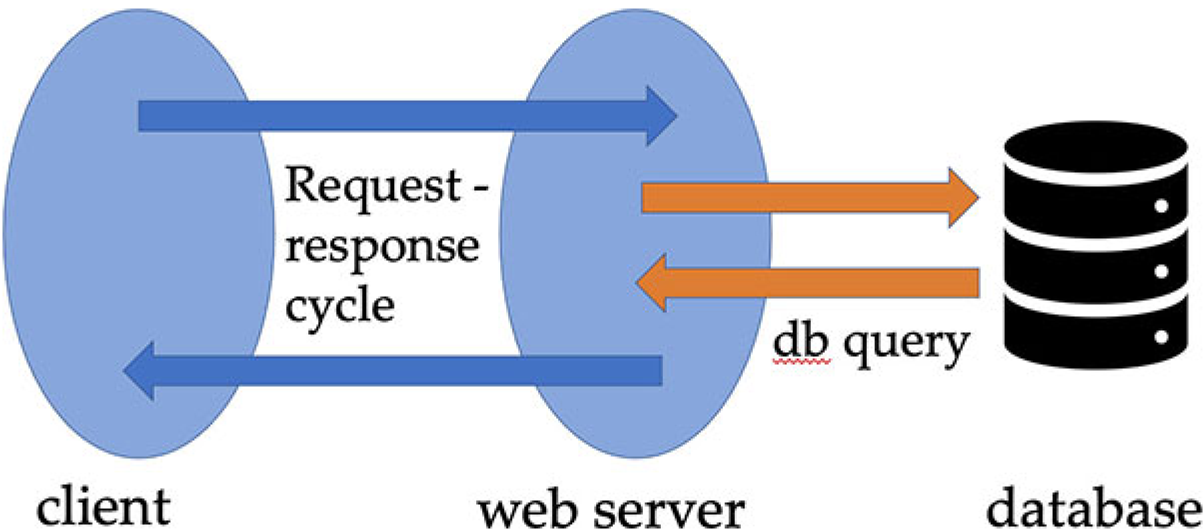

Figure 7.9: *Web server architecture with database backend*

A side effect of this approach is that the web server application builds a dependency on an external service, such as a database. This means the life

cycle operations of the server (like developing, testing, deploying, running, and upgrading) must also consider the presence and interplay of the database service now.

Design considerations

Now, let's examine various enterprise aspects of the application and see how those influence the application design.

Functional

When interacting with a database service, the key consideration is how the interaction will be carried out—exactly like we discussed in the case of internal and external services—by embedding the database calls between the request-response cycle, potentially passing the request context to the call after properly sanitizing the data. In addition, the database service will have a calling semantics that is not native to our application, in which case the caller will need to have an 'adaptor', 'connector', or a 'database client' that manages the communication with the database service.

The following code illustrates making a database query in the request-response cycle:

```
1. const h = require('http')
2. const d = require('mydbclient')
3. const u = 'http://mydb.remote:9000/u=user&p=password'
4. const s = h.createServer((q, r) => {
5.   const c = d.connect(u)
6.   c.query({name: 'foo'}, (m) => {
7.     let data = '';
8.     m.on('data', (d) => {
9.       data += d
10.     })
11.     m.on('end', () => {
12.       r.end(data.toString())
13.     })
```

```
14.   })
15. })
16. s.listen(12000)
```

Another major consideration is what to store in the database. The general answer is, data that needs to be persisted. More specifically, if the server crashes, we want the data that was already retrieved, computed, processed, and accumulated to be available without the need to repeat these processes. The main data elements typically stored in a database in the context of a web server are:

- **Template data (also called bootstrap data)**: Data that defines axioms/ground-truths around the server application's business logic. Examples are dictionary of terms, common messages, policy data, and so on.
- **User data (also called form data)**: Data that is retrieved, computed, processed, filtered, pruned, and transformed from the user forms as part of client requests. Examples are customer profile and account details.
- **Session data**: Data that holds the context of ongoing user transactions. We learned about sessions in *Chapter 6, Major Web Server Components*.
- **Derived data**: Data that is either an aggregation or higher-order derivative of one or more of the above-mentioned data. Examples are count of user requests in a 24-hour period, the total amount transacted from a specific account, and so on.

With the knowledge that the web server can be decomposed into microservices and the server can store business data into databases, a natural question arises: will there be individual databases for individual microservices, or a single, central database for all the microservices?

It depends. It is desirable to have individual database instances for microservices in most cases, but this imposes its own side effects. What if the data stored in individual databases has redundancy, relations, priority orders, and such? We are dealing with a single application logically, so it is natural that the discrete databases contain fragments of data in an

interleaved manner from the application's perspective, though coherent from individual microservice's perspective.

The database design needs to take care of such anomalies. If aggregations, normalizations, or synchronizations are required, we need to define events, triggers, and timers for initiating such activity. If the number of anomalies is very large, we will need to redesign our database design. The independent database model works well if such things are not present and the data for each service is mutually exclusive.

Performance

An additional set of network transaction is involved between the original client-server interaction, so the same performance degradation that we saw in the case of an internal service applies here as well. In addition, external services are almost always hosted on different servers, so network latency can be more than that of internal services. There are a number of things that we can take care of at both the database end and the web server end considering these factors and to offset the lost performance.

Database end

- Ensure that the database and the documents are aligned with the type of data stored—structured versus unstructured, and so on.
- Ensure that the documents are designed for easy retrieval for common queries.
- Ensure that the documents are aligned with the existing and possible future query types.

Web server end

- Ensure the implementation of connection pooling (reusing of connection) as opposed to making fresh connections every time (saves network latency by large).
- Ensure that multiple parts of the server do not cause contention at the database side. If there are queries that cause bottlenecks, either implement enough isolation between the, or consider coalescing the queries into one.

Reliability

The same reliability measures that we adopted for internal and external services apply as is to database services as well, in addition to the network outage possibilities in the internet – as our database service sits across the network on the internet.

In the case of internal/external services, we discussed how we want to manage the service failures and how we want to gracefully respond to the user through the response. What is the case when the database is down? Can we think of a similar strategy? No. The database holds vital business data that is essential for the server to function well (including session data), so it is neither feasible nor meaningful to continue accepting new connections. For all practical purposes, we should treat the database as a vital part of our server component itself, although it is discretely situated across the network as a module.

Security

Given that the communication between the web server and the database service is now happening across the network, and a person skilled in the art can very well speculate what would be typically stored in a database, the communication between the server and the database becomes a security topic. A few important security measures are mentioned as follows:

- **Database level**: Design and organize the data in such a manner that different users have different and matching/deserving levels of authorization/access to the data.
- **Connection level**: Ensure that opening up a request with the database is allowed only upon logged in connections, and that the connection parameters are strongly typed.
- **Transport level**: Ensure that the data—inward and outward—is encrypted to prevent theft.
- **Query level**: As we studied in *Chapter 6, Major Web Server Components*, database queries crafted through untrusted user input can potentially harm the database. So, ensure that the query strings are properly validated and sanitized before relaying them to the database.

Serviceability

Just like any other external service, we need to integrate the serviceability features of both the web server and the database and make the feature seamless to improve problem determination experience. However, by isolating part of the application program (code and data) and implementing well-defined control and access points to manipulate that part of the program (database), our overall serviceability confidence increases instead of deteriorating.

Conclusion

In this chapter, we understood the general objective and design of leveraging backend services in our web application. We looked at three different flavors of backend service types and the fundamental design considerations of interacting with each of those services. This helped us easily develop a server with one or more such backend services, architect newer interactions, extend the existing interaction, or optimize service invocations for improved performance, reliability, security, or serviceability.

In the next chapter, we will look at the last set of missing pieces of our website—common requirements of a website (front-end rendering) and how the pages and forms can implement some of those common features. We will see how large amounts of site data can be rendered for better consumption (pagination, search, and filtering), and we will also cover implementing authentication and authorization and other common requirements. Technically, that will wrap up the constituent elements that make up a website.

CHAPTER 8
Implementing Common Website Features

After learning about web server concepts, we went on to cover all the backend components: the web server components, the various middleware components that support the server for its functionality, and backend components that the server leverages for extended capabilities. The only remaining piece in the bigger picture of the web architecture is frontend elements. In this chapter, we will look at the common requirements of a website (frontend rendering) and how the pages and forms can implement some of those common features. We will look at the considerations of web page constitution based on different use cases and how special components meet those requirements in page design. Technically, that will complete the coverage of constituent elements that make up a website.

Structure

In this chapter, we will cover the following topics:

- Website – design considerations
- Website – elements and components
- Website – advanced features

Objective

After studying this chapter, you will be able to understand the general objective and design of the frontend components in your web application. The discussion will include user experience around the frontend design, constituent elements that shape user experience, and some advanced features that make up modern web applications' frontends. These learnings will help us build a scalable, responsive, and modern website. More

importantly, the selection of items, the order in which they're picked up, and the depth of our discussion of those components will help us rearchitect the components in a typical development process based on changing needs.

History of web pages

The Web started in the 1990s with simple, distributed servers serving static content. Then, HTTP specification was defined. The need for small-scale validation and animation of the pages gave raise to JavaScript. As the use cases of the Web widened, cookies were invented. Soon, many other languages (such as Java) started implementing client-side frameworks (such as Applets) that run in conjunction with HTML pages. Consequently, the HTTP specification was refined and upgraded.

The early 2000s brought in an increase in use cases such as interactive pages and that led to the evolution of Asynchronous JavaScript And XML (AJAX), which empowered parts of the pages to dynamically and asynchronously interact with the server and update part of the views, bringing high level of interactions without wasting network bandwidth while also enhancing user experience.

Cascading Style Sheets (**CSS**) that centrally define and control the look and feel of pages, web workers that allow script execution in background threads, rich multimedia support, and the ability to embed third-party content added exponential value to web pages. At this point, web pages were able to functionally deliver superior value than their thick client counterparts (desktop applications).

In the early 2010s, mobile devices became prominent consumers of web pages, disruptively changing the user experience and thereby, user interface considerations. With that, the current web page design became an extremely complex set of intents that must cater to the needs of multiple types of use cases and users.

In this chapter, we will examine the expectations of a typical current user in terms of user experience, elements in a website that meet such expectations, and design considerations around these elements.

Website – design considerations

In general terms, user experience refers to the degree of intuitiveness, usability, ease of use, and efficiency of a product. So, developing a product with satisfactory user experience requires us to understand the type of users, user behavior, user expectations, and their abilities and weaknesses with respect to product consumption.

Relating this definition to a website, it boils down to the perception of a web user on how aligned the website is for their needs and how compelling it is for them to retain its use in comparison with other websites that present similar offerings. What are those compelling reasons? Let's look at each of those in detail.

The following diagram shows the main elements of user experience on the Web:

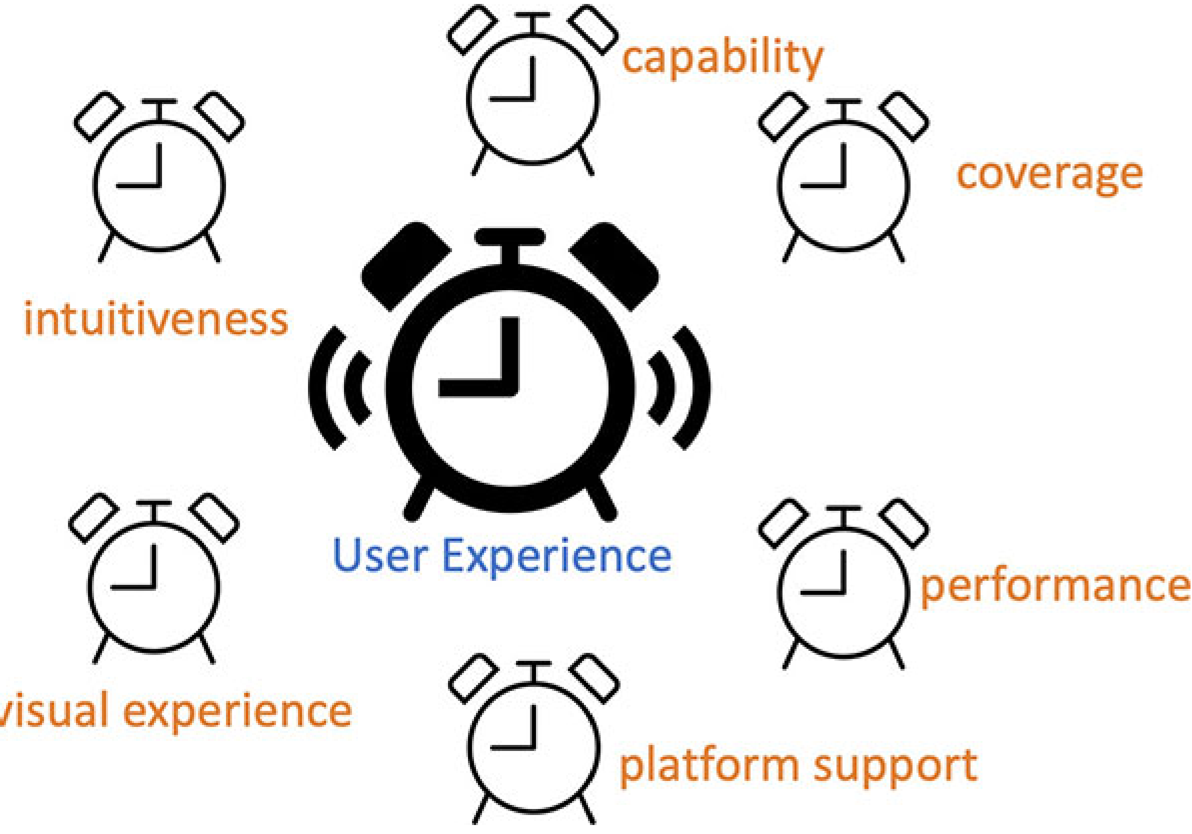

Figure 8.1: *Elements of user experience*

Usability

In the usability section, we look at some of the considerations that directly relate to the usability of web pages. These very fundamental considerations

for a user and create a strong impression about the website.

Registered and unregistered views

How does it feel when a user opens a website and they get a huge login form as a pre-requisite for access to its content? Obviously, this is a visible inconvenience to the user. A website, whatever important functionality it offers, should have unregistered views. If the content is confidential and needs a valid user context and account to access it, the site could provide high-level narration on the objective, the nature of content inside, how it is useful, and what types of users the website is relevant for.

The following figure illustrates a bad user experience on the home page when everything is concealed through a login/registration flow.

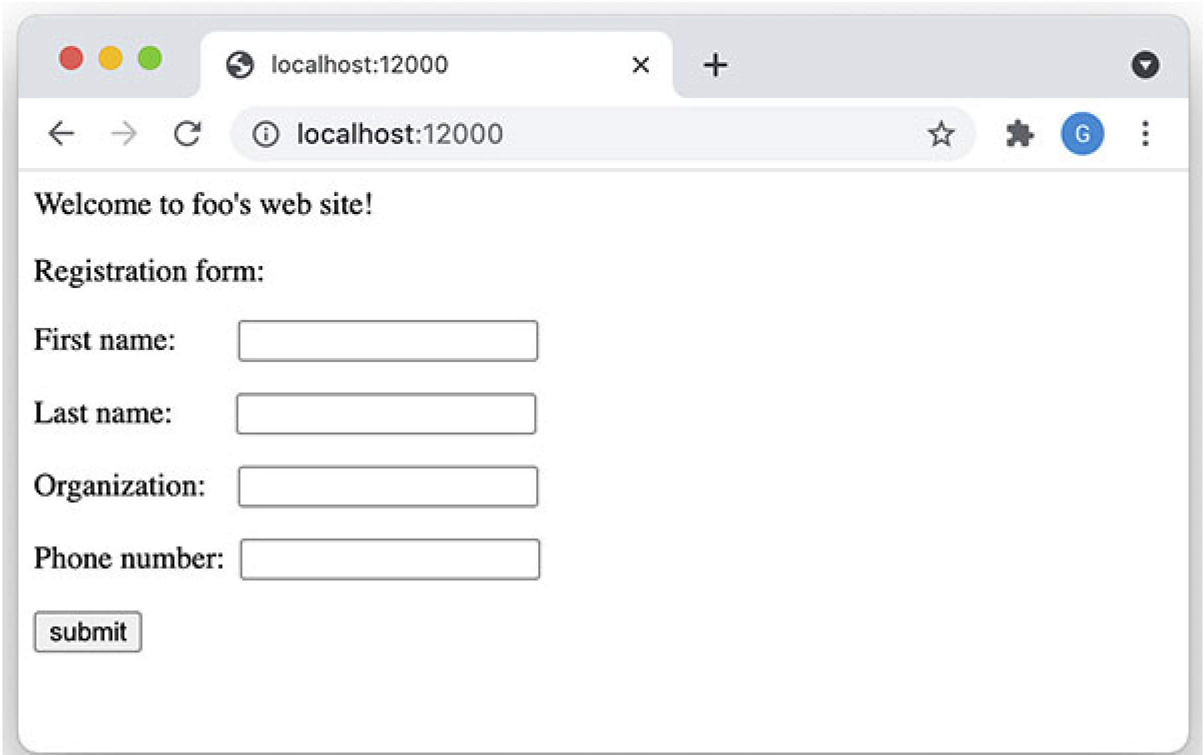

Figure 8.2: *A home page with no unregistered view*

On the other hand, the following figure shows how this can be improved by bifurcating between registered and unregistered views:

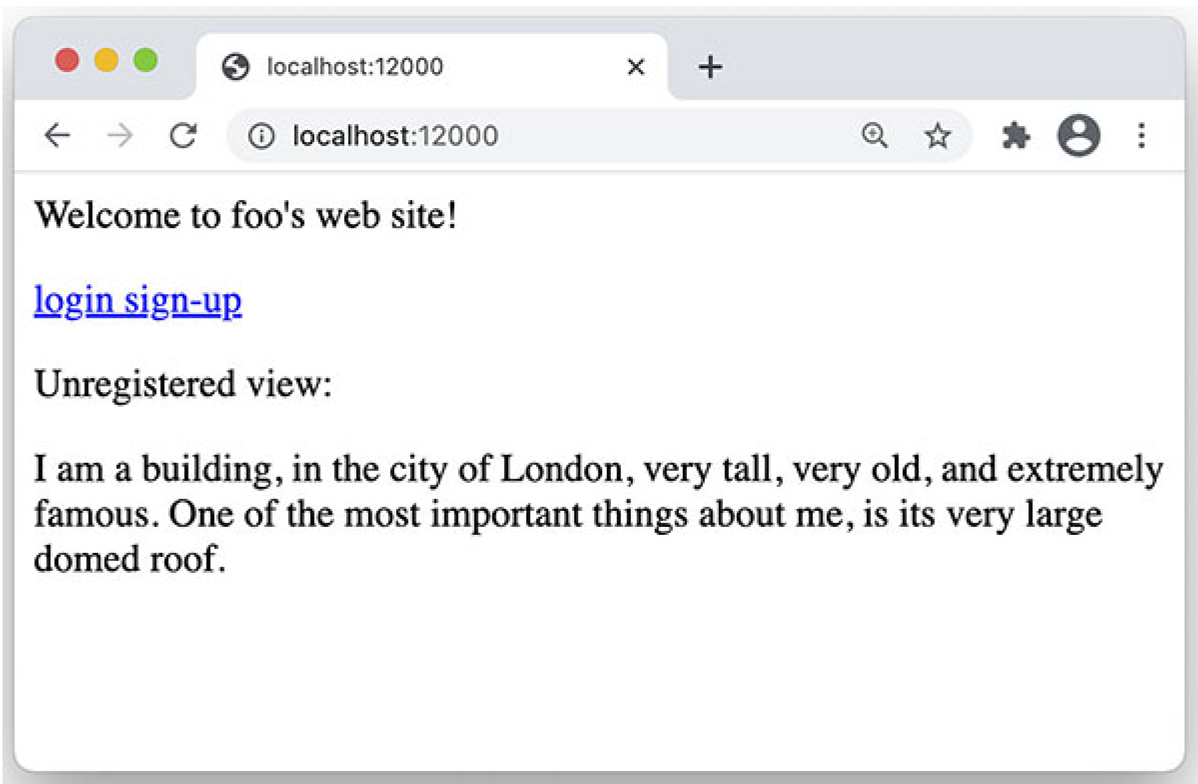

Figure 8.3: *A home page with unregistered views*

Navigation bar

Most websites will have multiple pages, organized hierarchically. An old design is to provide navigation buttons (previous, next, home, and so on) for the user to move from one page to another. This roughly means the user needs to keep track of their navigation history. A modern alternative is to provide a navigation bar that is available in all pages so that the user is free to move from any page to any other, and it also spares them from 'remembering' the navigation history.

Here's a screenshot with a few menu items and a navigation bar that appears commonly for all pages:

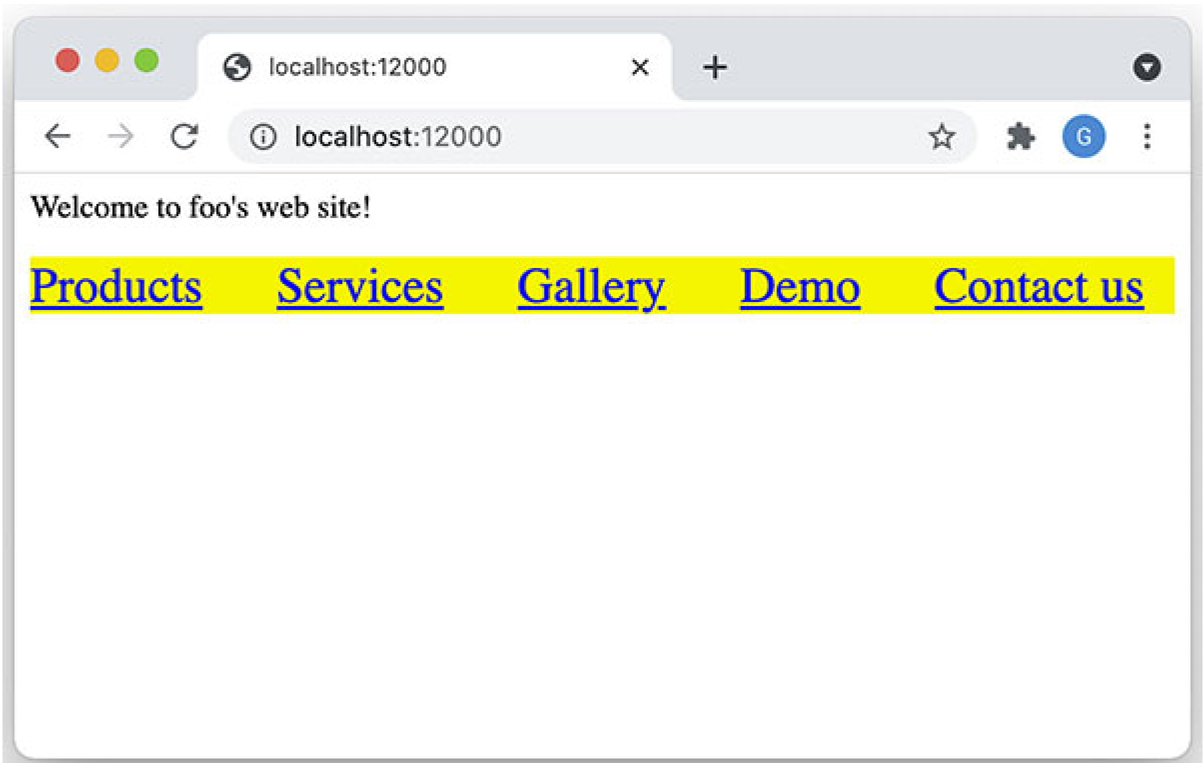

Figure 8.4: *A web page with navigation bar*

Articles

Articles constitute the main trunk of a page. Rather than spreading it in the entire page, an article placed at the center of the page with sufficient focus significantly improves the page's usability. The following figure shows an article section:

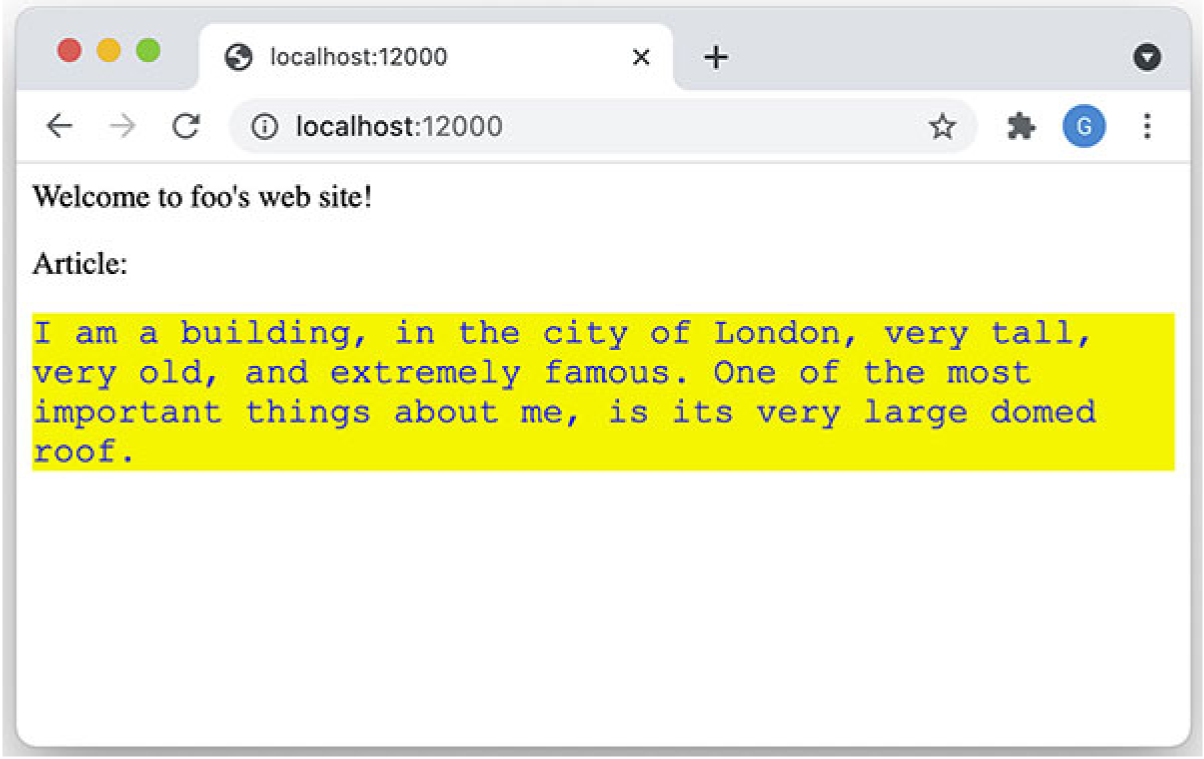

Figure 8.5: *A web page with article tag*

Headers and footers

A page header is a horizontal bar on top of the page. This section ideally contains website logo, common website information, and any press releases.

The following diagram shows how a page header may look:

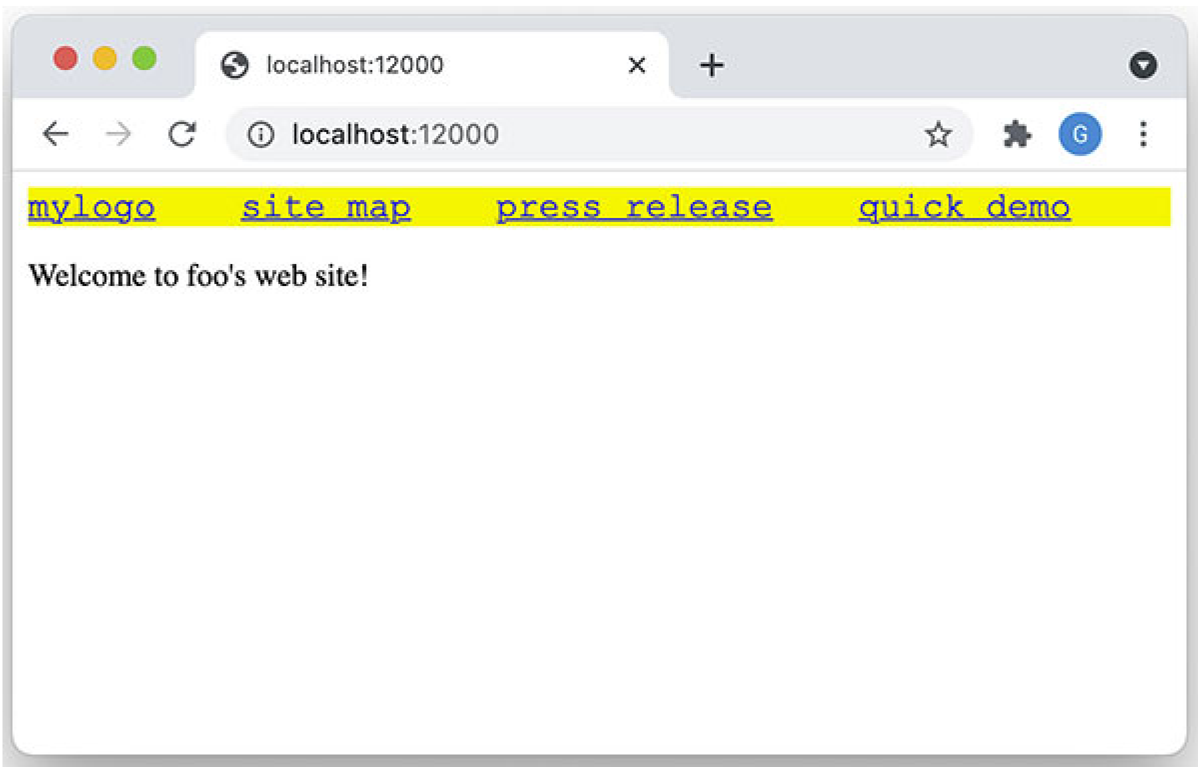

Figure 8.6: *A web page with header*

A page footer is also a horizontal bar, but at the bottom of the page. This Copyright information and contact details are generally placed in the footer section. The following screenshot shows what a page footer may look like:

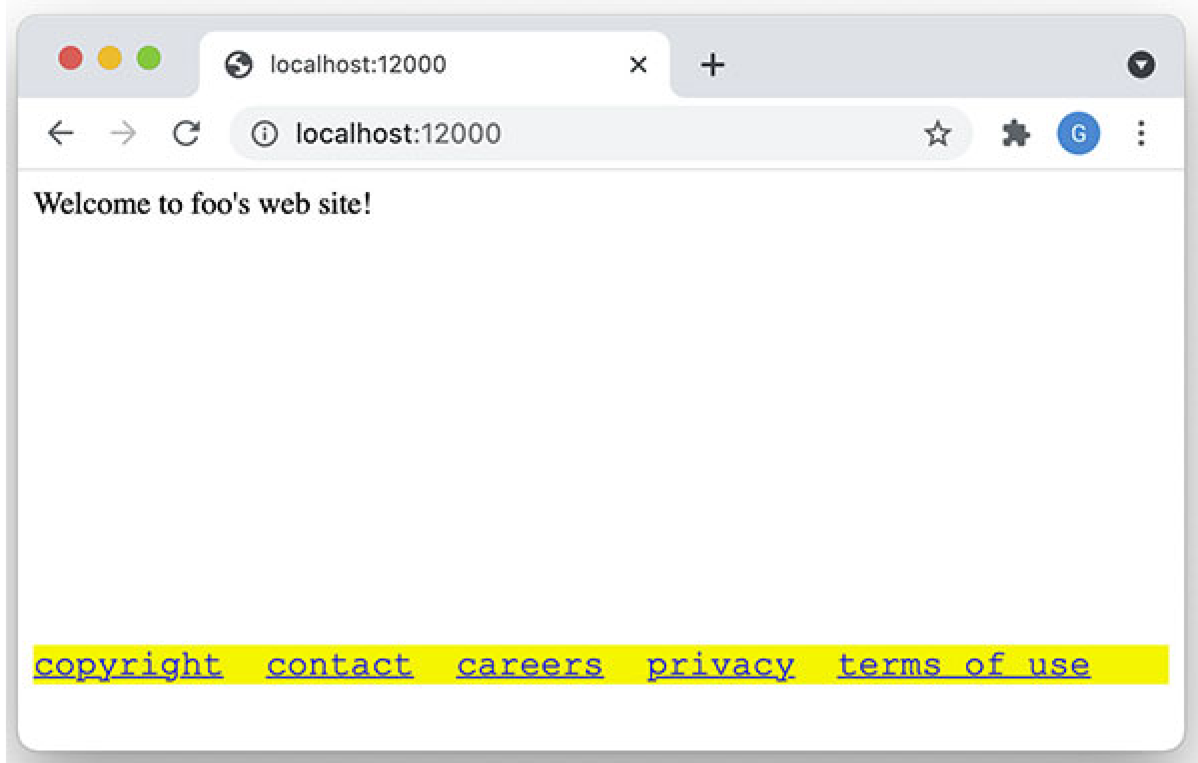

Figure 8.7: *A web page with footer*

Forms

A form is an essential part of many websites. An organized web form helps collect related information from a user for processing at the backend. The most common form types are registration and profile forms.

The following screenshot illustrates a simple web form:

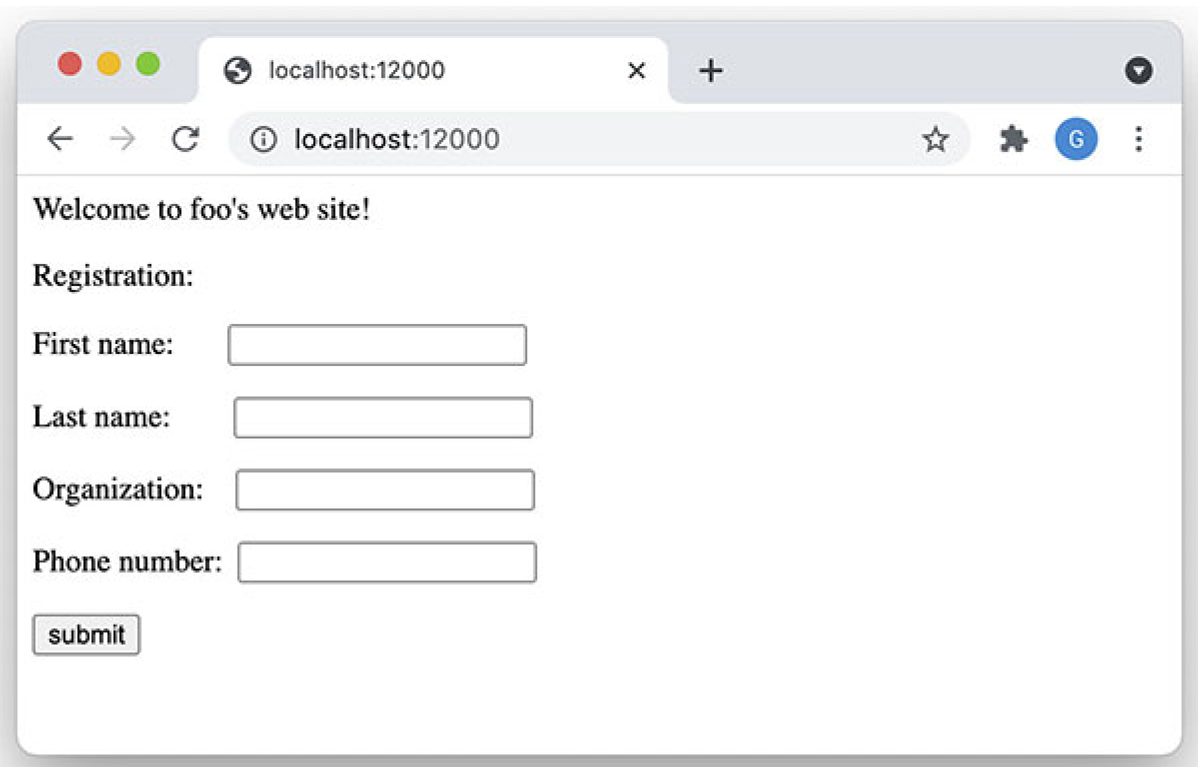

Figure 8.8: *A web page with a form*

Search option

A good site will have a search bar that provides a custom search function confined to the website content. This is useful when the site provides a relatively large number of features and content and some of the items are not directly discoverable.

The following screenshot shows a page that embeds a search option:

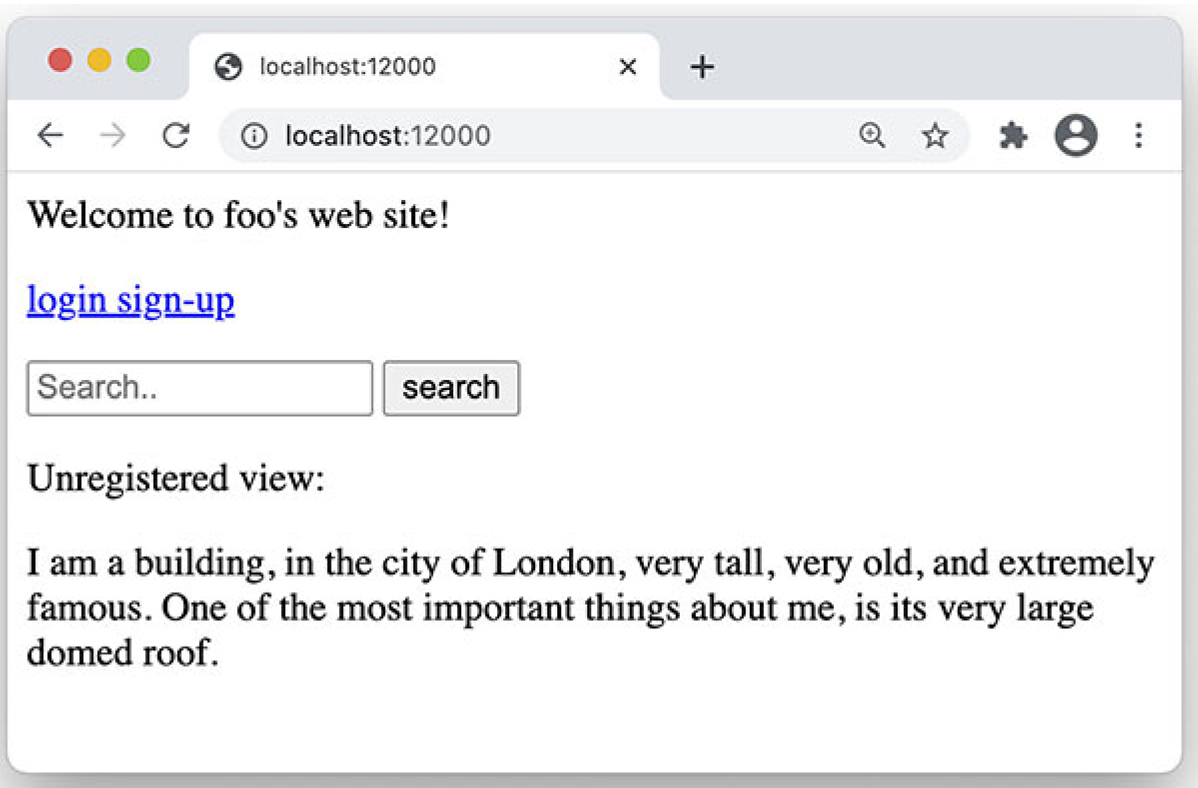

Figure 8.9: *A web page with a search bar*

Feedback forms

Collecting user feedback directly is one of the easiest ways to improve user experience, so it is highly desirable to have a feedback form where a user can enter their overall experience and any improvement suggestions.

The following screenshot illustrates a typical feedback form:

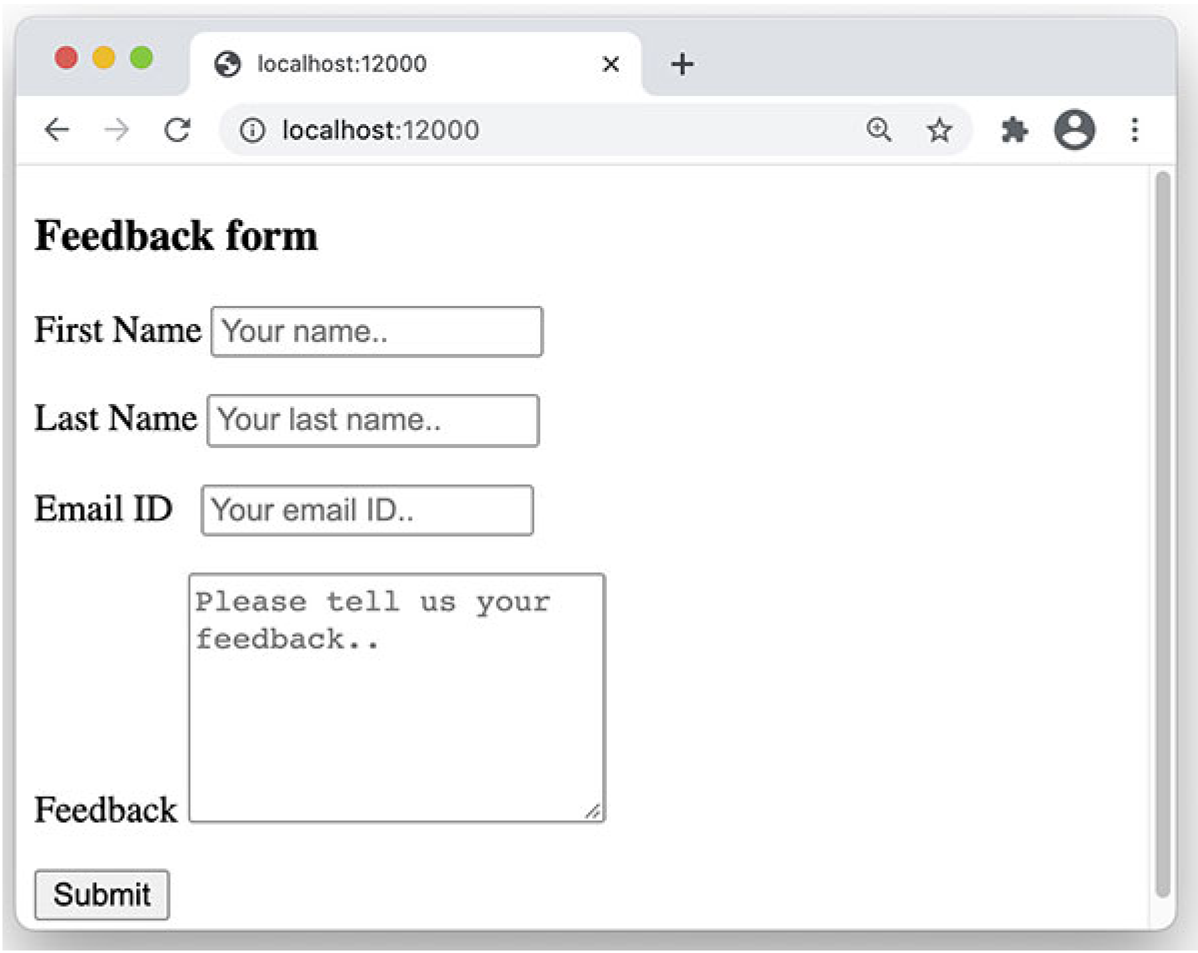

Figure 8.10: *A web page with feedback form*

Shopping carts

Wherever applicable, a cart feature provides users with the ability to 'add' commodities to a tray so that actions like purchases can be performed in aggregation, instead of needing to perform common transactions repeatedly.

The following screenshot illustrates a typical usage of this pattern:

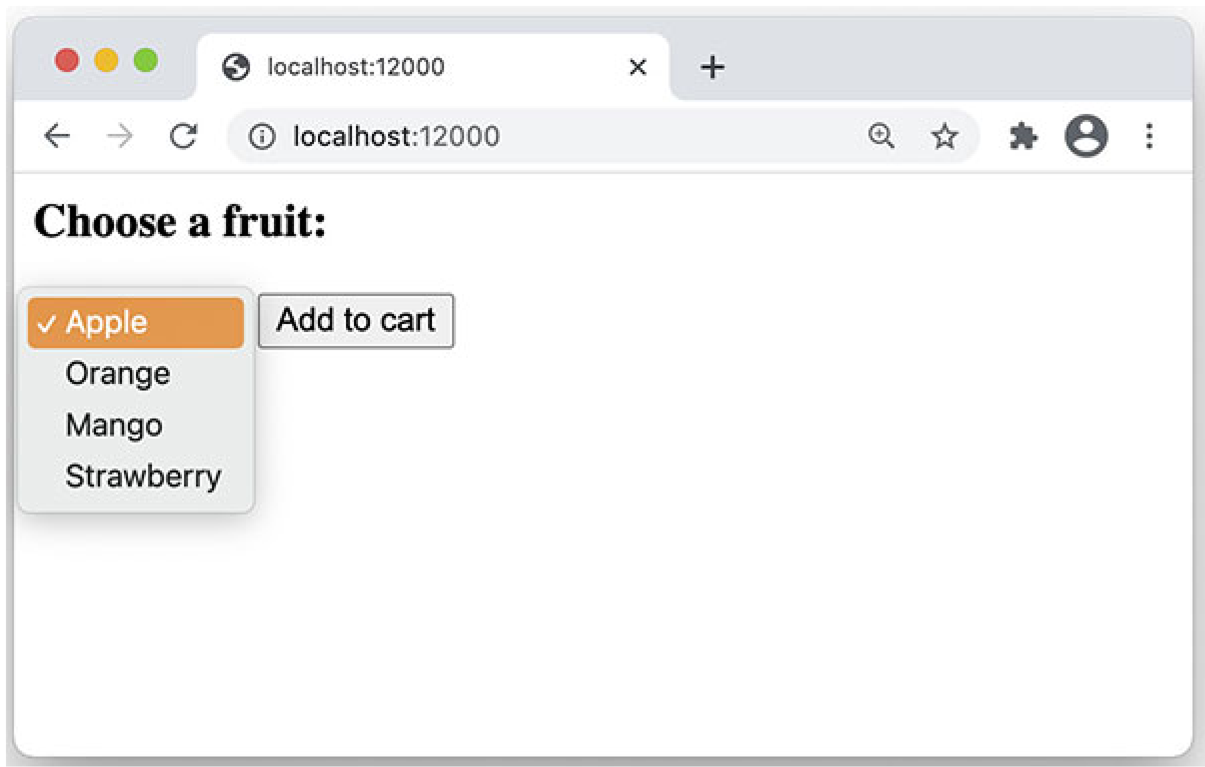

Figure 8.11: *A web page with shopping cart*

Payment options

In a web application, a payment feature provides users with the ability to 'pay' for an item or a service that the web application offers, without leaving the website. Keep in mind that depending on an external vendor/website to perform the payment on behalf of the current website will lead to poor user experience.

The following screenshot illustrates a payment form:

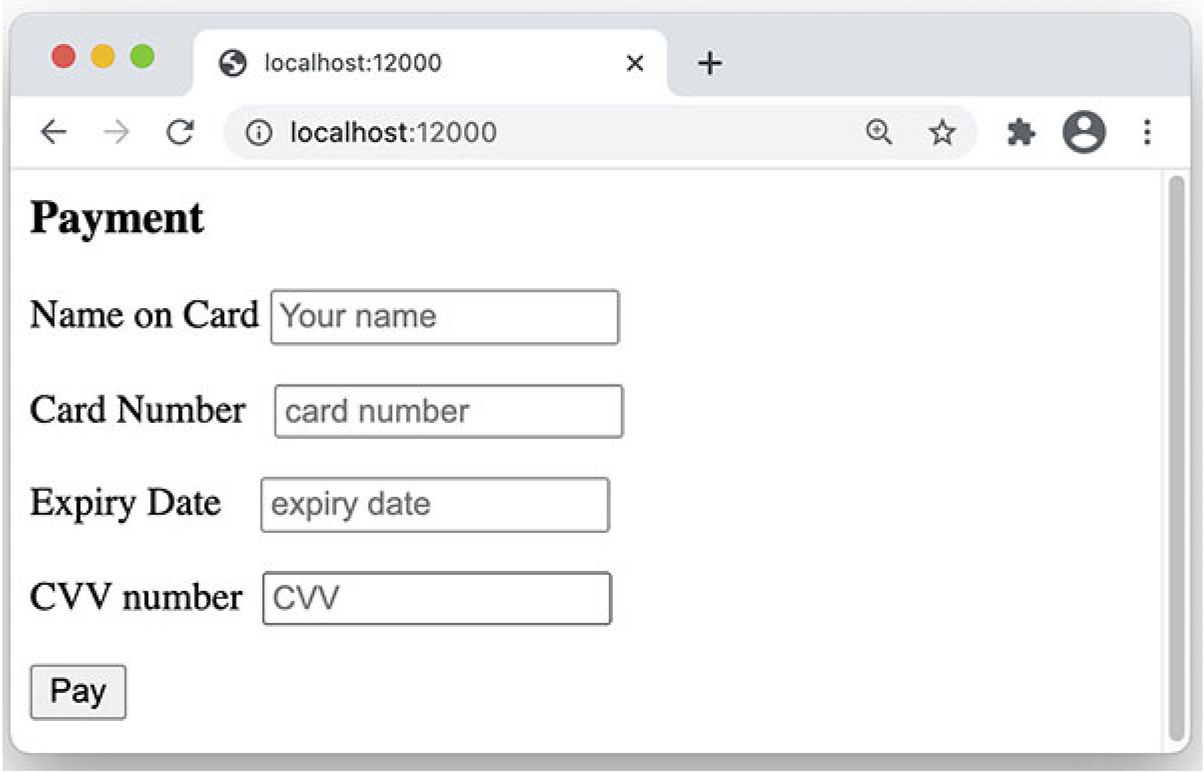

Figure 8.12: *A web page with payment option*

Ease of use

In this section, we will look at some of the non-functional aspects that are directly related to user experience.

Consistent pages

The following are the traits of consistent pages for improved user experience:

- Having a similar look and feel in related pages
- The placement of controls and components in similar coordinates
- Uniform appearance of controls and components
- Behavior of controls and components in an expected manner

The following screenshot illustrates an inconsistent set of pages:

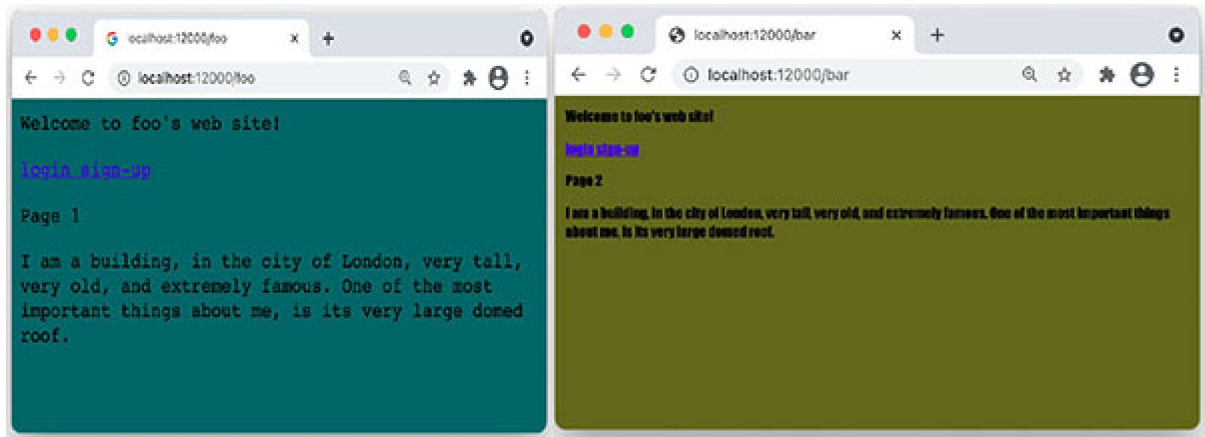

Figure 8.13: *Inconsistent web pages*

On the other hand, the following screenshot shows web pages made consistent:

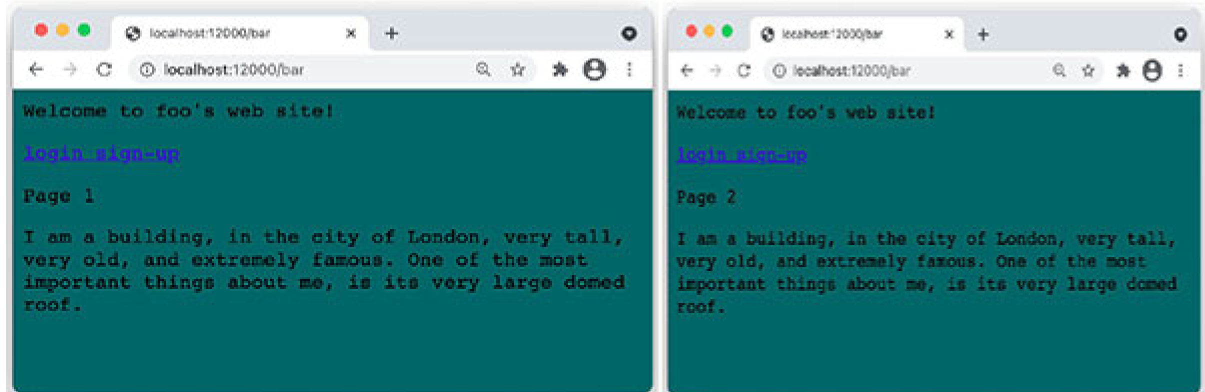

Figure 8.14: *Consistent web pages*

Question: Which framework/technology provides the capability to implement consistent look and feel to a bunch of web pages?

Short pages

Ideally, you should try to fit the page in one screen and avoid the need to scroll as much as possible. But of course, at times, a coherent set of content may have to be kept on a single page, irrespective of its volume. In such cases, it is worthwhile to look at meaningful segregations for the content so that you can still split it.

The following screenshot shows a short web page, which is easy to consume:

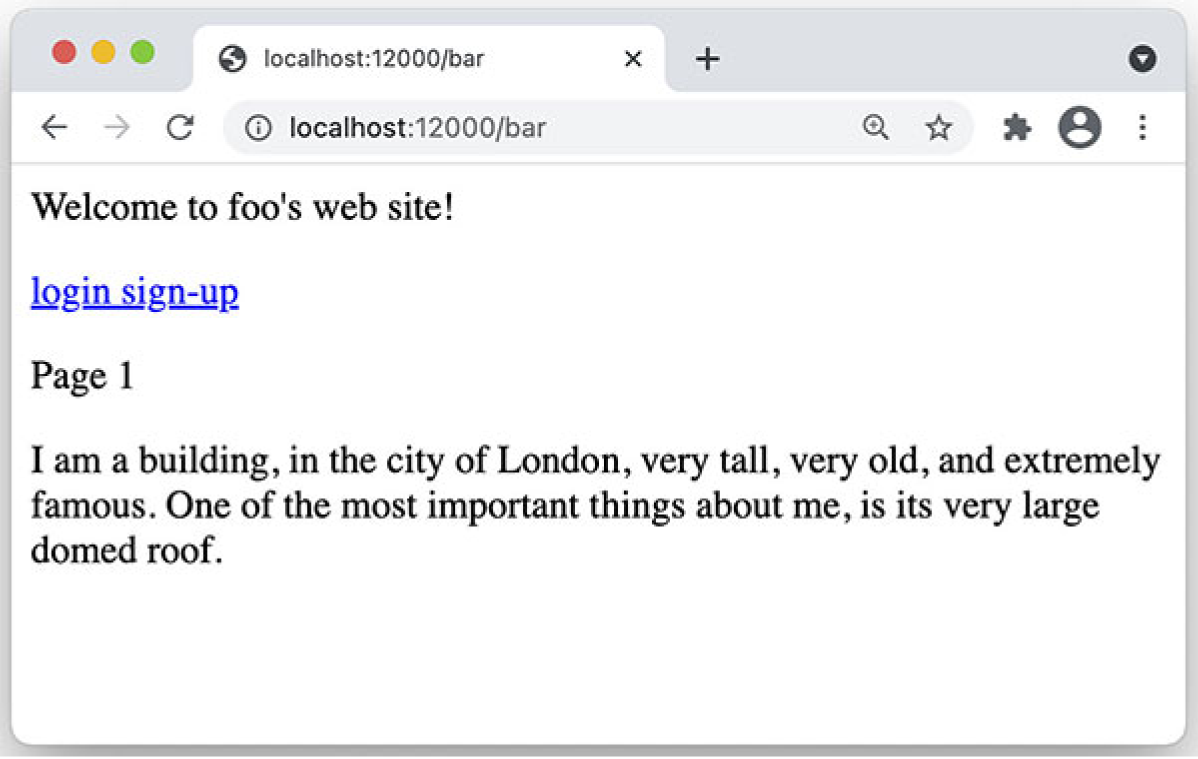

Figure 8.15: *A short web page*

Short forms

Forms are very important. They switch user effort, from simple consumption and navigation to careful feeding of information. Some of the features that improve user experience with forms are:

- Collect minimal information (which is required)
- Follow a logical ordering of the form elements
- Label the input fields appropriately
- Provide additional help text for ambiguous fields, providing default values wherever possible, auto-completion based on past interaction, or other analytical means

The following screenshot shows a short web form:

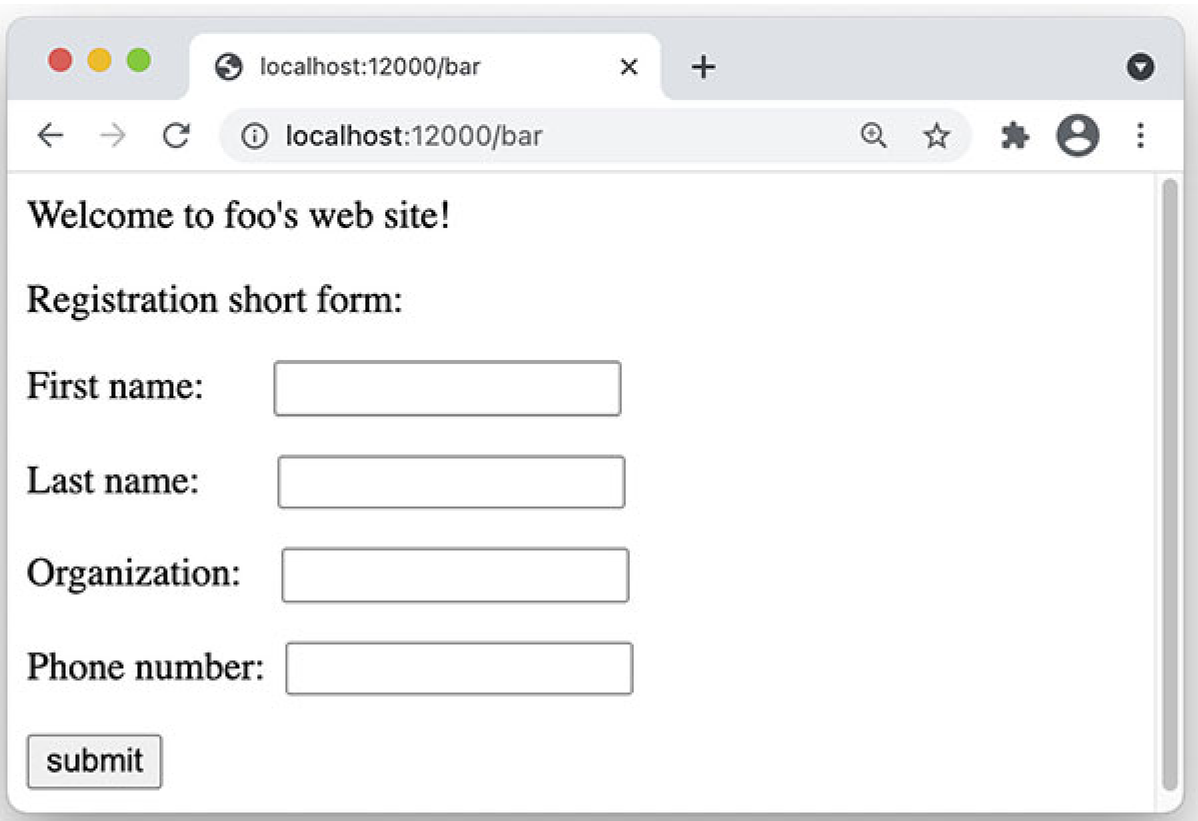

Figure 8.16: *A short web form*

Question: For the previous section (short pages), we said reduce the content such that it fits in one page. Is there a difference between short pages and short forms in terms of usability and ease of use? For example, what happens if you really need to capture critical information that is not fitting in a single page?

Home page

In the early days of web evolution, a site map feature helped users see where they have reached in the navigation and how to get out of return. Modern websites either have a navigation bar that is available on all the pages or a home button that is available everywhere so that one can easily return to the first page.

The following screenshot shows a link to the home page that is made available across the site:

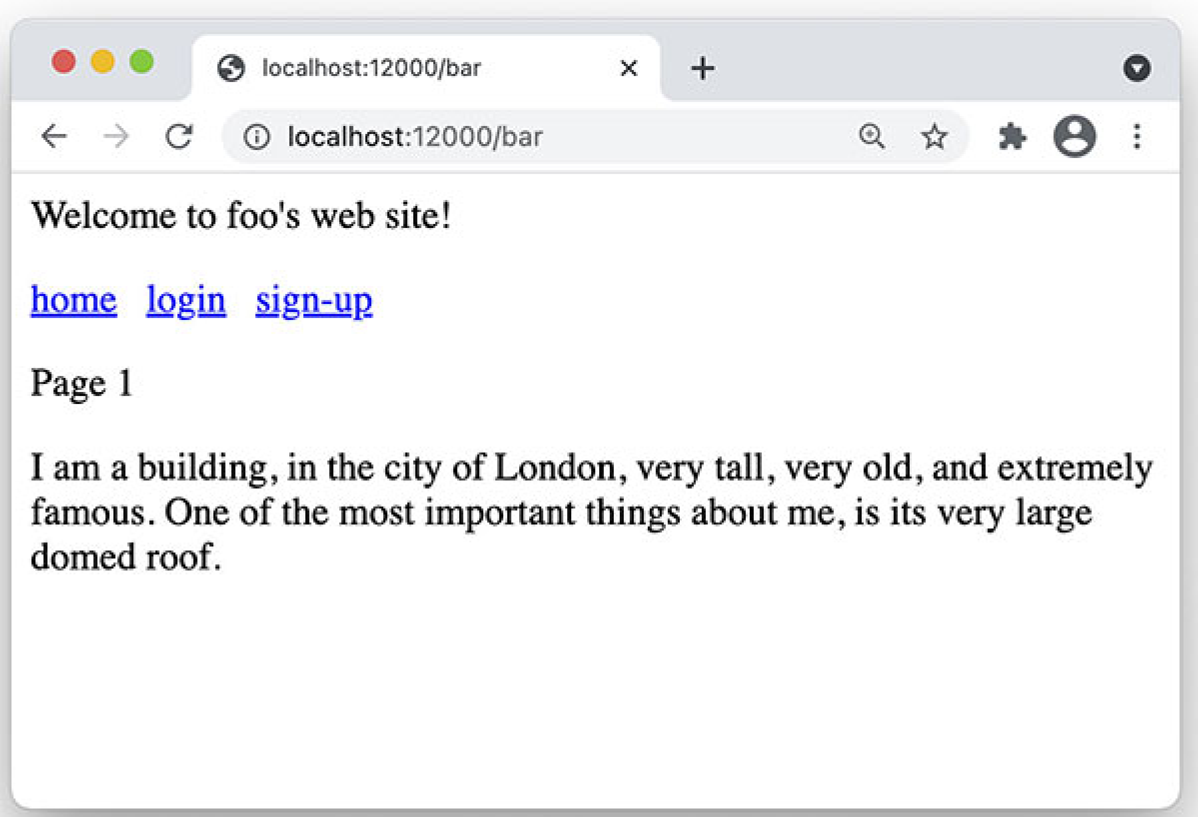

Figure 8.17: *A web page with home page link*

Responsive web forms

Web content is not only consumed through browsers but also through a variety of mobile devices, tablets, and other smart devices like television, so the content should provide a uniform experience through all forms of interaction. To achieve this, the page or form should adjust its dimensions and scales based on the type of the target device.

The following screenshot shows a responsive web page that has a similar appearance in all forms of consuming devices:

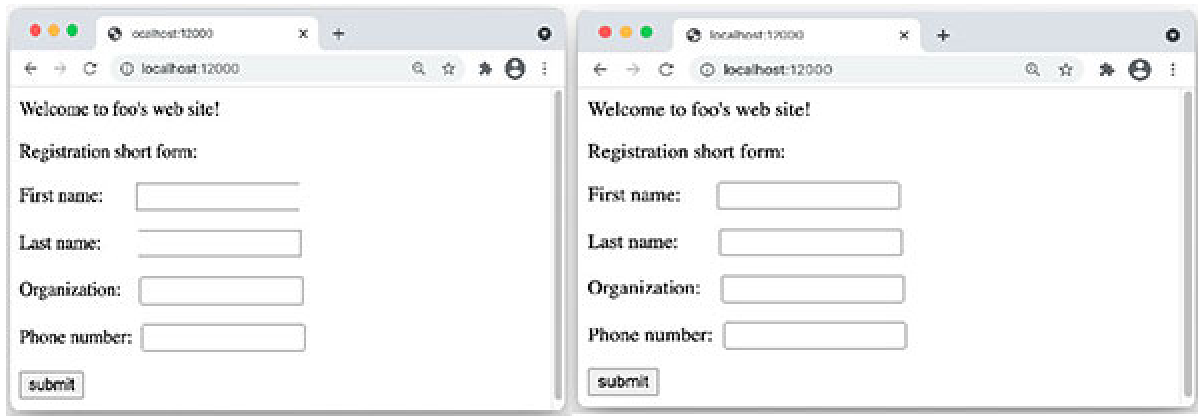

Figure 8.18: *A responsive web page with consistent look and feel and behavior*

Note: One of the primary considerations for web pages designed to be rendered in mobile devices is the inefficiency with the keyboard. As most mobile keyboards are software simulations as opposed to physical keys, the typing is more difficult than on their desktop counterparts. Its implication on web page design is that the interaction should be such that user input has to be minimized or should boil down to clicks and selections as opposed to bare typing.

In summary, we looked at a number of factors that shape user experience of the web content consumption, in terms of usability as well as ease of use. These are generic concepts that apply across the World Wide Web and have nothing to do with a specific development platform like Node.js. Now, let's look at some of the actual elements and components that constitute a web page and those that contribute to achieving the said user experience.

Website – elements and components

So far, we talked about the user experience and design aspects of websites in general. Now, let's the main elements of a single web page that constitute websites. We classify these based on multiple criteria, such as language, elements, and components.

By language

In this section, we look at the languages commonly used for rendering a web page. We also illustrate how they interact with each other, manifesting

perfect polyglot behavior by the client devices.

HTML

Hyper Text Markup Language (**HTML**) defines the fundamental structural semantics for web document. It has a rich set of syntax to represent page appearance and attributes to cover a wide variety of features. This is supported by all browsers and client devices that consume web content.

The following code snippet shows the shortest HTML syntax:

```
1. <html> hello html</html>
```

Question: Assume that we have browsers that do not support HTML. How do you think the client-side rendition can be implemented with the help of another language, say C?

CSS

Cascading Style Sheets (**CSS**) defines the presentation styles for plurality of pages for a website. It separates web content from its format, and thereby reduces the complexity of managing web content. The main formatting that comes under the purview of CSS are colors, layouts, and fonts.

The following code snippet shows an inline style sheet embedded in HTML:

```
1. <html>
2. <body>
3. <p style="color:blue;">hello css</p>
4. </body>
5. </html>
```

Note: There are semantics defined in CSS that help us refer to part of the HTML document and apply styles selectively. This design helps improve the separation of concerns, isolating the core component definitions from how they should appear.

JavaScript

JavaScript is an event-driven, functional, and imperative programming language with APIs for general purpose programming. In the client's execution environment, it has constructs to access the components of the web page it is a part of.

The following code snippet shows an inline JavaScript embedded in HTML:

```
1. <html>
2. <script> alert('hello JavaScript!') </script>
3. </html>
```

> **Note:** This aspect of the web frontend – the ability to embed JavaScript and thereby manipulate the content in markup format asynchronously and dynamically, with the help of the same programming language that we used in the backend (Node.js), is by far the most important reason for the proliferation of JavaScript in the web workloads.

In summary, among these three languages, HTML acts as the main trunk of the web page and focuses on organizing the content, CSS focusses on providing common style and format to the web page, and JavaScript provides dynamic rendition by exhibiting multi-tasking and asynchronous programming.

By elements

In this section, we will look at various programming abstractions and interfaces to work with a web page, at the front end. This includes plain HTML as well as JavaScript programming interfaces that work in conjunction with the HTML elements.

DOM

Document Object Model (**DOM**) is an abstraction of web content. The model allows programmatic access to the web content, which is organized as a tree structure and available as a composite object in the language through which the content is accessed. This abstraction helps us modify the

web content after those (the individual components of the web content) have been rendered/scheduled for rendition and bring dynamic content to the page by installing events and event handlers specific to parts of the document.

The following code snippet shows how JavaScript can access and modify the HTML components, leveraging the Documented Object Model abstraction:

```
1. <html>
2. <p>
3. <body>
4. <div>
5. <input id="foo" placeholder="foo"></input>
6. </div>
7. <script>
8. document.getElementById('foo').placeholder = 'bar'
9. </script>
10. <p>
11. </html>
```

After running that, we see that the original value of the input placeholder ‘foo’ has been changed by the script to ‘bar’, which is showcased as follows:

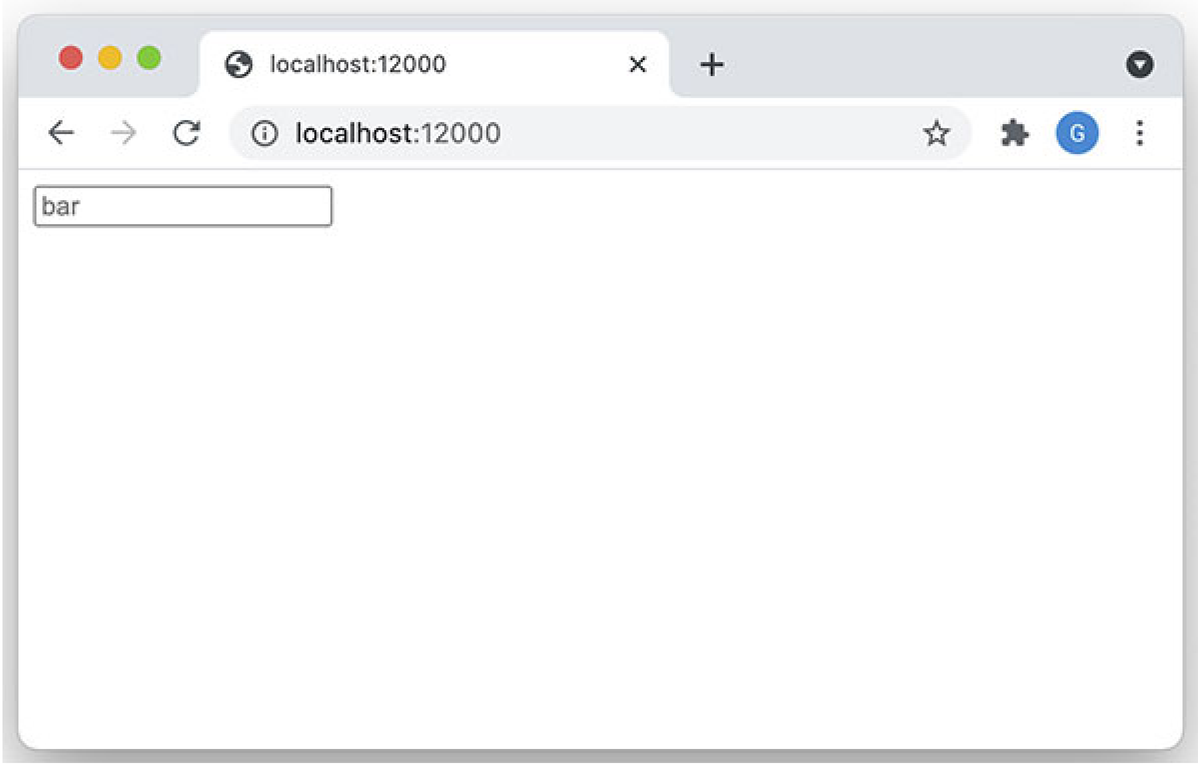

Figure 8.19: *A web page with JS manipulation*

Note: An additional advantage of using JavaScript in web pages is the ability to improve client interaction more efficiently, from simple validations to imparting complex event-driven behavior to elements in the page, leading to a perfect complementation of HTML's capabilities.

Assignment: Demonstrate the local management capabilities of JavaScript by adding an event handler function for an HTML element and explain how part of the server functions are delegated to the client side, thereby reducing network bandwidth and improving performance.

XMLHttpRequest

Despite its somewhat strange name, this is another abstraction for direct HTTP interaction with a backend server. But why another networking abstraction when the browser has native interfaces to deal with the server either through its address bar or through special HTML elements and attributes? Those interfaces affect the current page as a whole. For example, a URL request through the browser will invalidate the current page and

render the resulting content from the server. On the other hand, the **XMLHttpRequest** APIs interact and obtain content from the backend, which can be used to 'hand-craft' the existing content (think about the DOM update explained in the previous section) and refresh part of the page. This enables the page to appear lively.

Think about a website that provides live scores of a match. The scoreboard refreshes itself at regular intervals, but the page doesn't reload as a whole, and the user doesn't refresh it either. For this, one implementation approach is for the JavaScript elements in the page to install a timer that kicks in at regular intervals, and issue a request with the server, fully transparent to the user.

The following code snippet is a server-side logic that handles **XMLHTTPRequest**:

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   if (q.url === '/bar') {
4.     return r.end('response to XMLHttpRequest!')
5.   }
6.   r.end(require('fs').readFileSync(process.argv[2]))
7. })
8. s.listen(12000)
```

The following code snippet (client-side) shows the usage of the **XMLHTTPRequest** API:

```
1. <html>
2. <body>
3. <p>
4. <div id='xhr'>
5. <button type='button' onclick='XHR()'>XHR</button>
6. </div>
7. <div id='non-xhr'>
```

```
8. I am a building, in the city of London, very tall, very old, and extremely famous. One of the most important things about me, is its very large domed roof.
9. </div>
10. <script>
11. function XHR() {
12.   const x = new XMLHttpRequest()
13.   x.onreadystatechange = function() {
14.     if (this.readyState == 4 && this.status == 200) {
15.                document.getElementById('xhr').innerHTML = this.responseText
16.     }
17.   }
18.   x.open('GET', '/bar', true)
19.   x.send()
20. }
21. </script>
22. </body>
23. </html>
```

The following screenshot shows how the initial rendition of the preceding code looks:

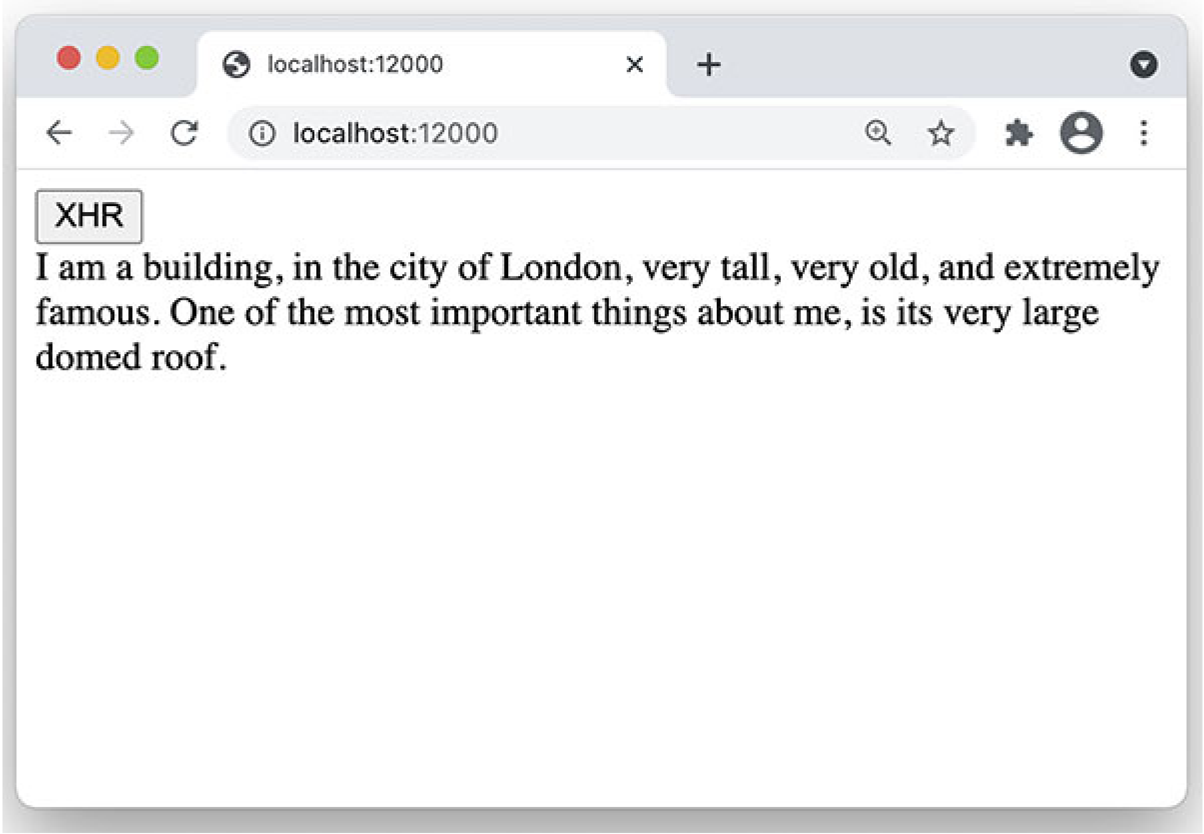

Figure 8.20: *A web page with elements controlled by XMLHTTPRequest*

Once the user clicks on the button, the API fetches new data from the server and renders that in the view, as follows:

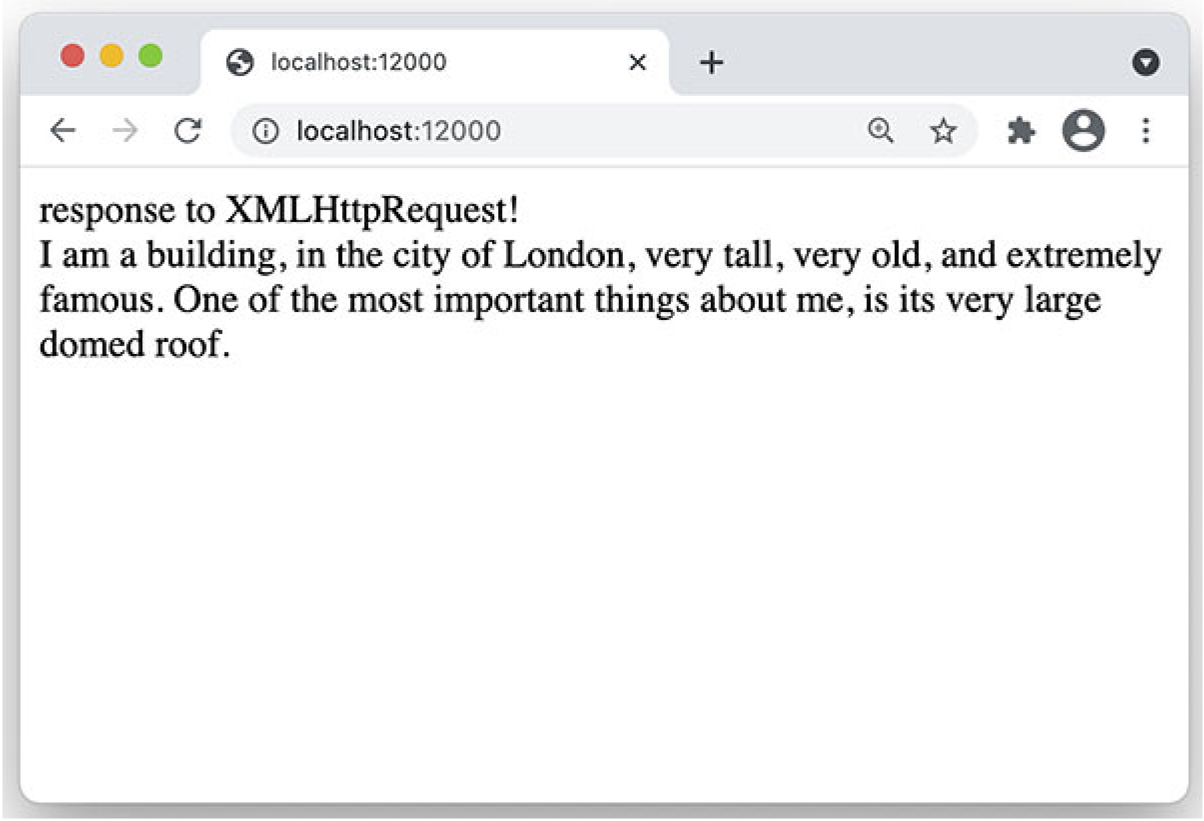

Figure 8.21: *A web page with the result of the XMLHTTPRequest action*

Question: While a client loads a web page, it has already established a connection with the server and obtained the content from it. When a component in the page subsequently issues an XMLHTTPRequest to the server, does it open a new connection with the server or reuse the existing connection established while loading the main page?

WebSocket

This is similar to **`XMLHttpRequest`** in terms of the model, and it is yet another network abstraction that enables a programming element in the web page to directly interact with the server through TCP protocol. The full-duplex nature of the TCP connection enables WebSockets to receive asynchronous events and data from the server. This leads to a feature wherein the server can proactively write to the client in response to a previously registered interest.

The following code shows a typical usage of web socket:

```
1. <html>
2. <body>
3. <input id="ws" type="text"></input>
4. <script>
5. const ws = new WebSocket('ws://localhost:12000')
6. ws.onopen = function() {
7.   alert('connecting')
8.     ws.send('hello')
9. }
10. ws.onmessage = function(e) {
11. document.getElementById('ws').value = e.data
12. }
13. </script>
14. </body>
15. </html>
```

Let's revisit the previous example where a live scoreboard needs to be implemented. With WebSockets, the timer loop in the client code is not required; instead, the server that intercepts the score change in the actual event (the game) is able to 'push' the new data to the client through the previously opened WebSocket. This significantly enhances programming efficiency.

Question: What are the similarities/differences between the use case of XMLHTTPRequest and WebSocket?

At some point around here, it is natural to think about the role of backend and frontend and the boundary between the two. With the previously-mentioned two features, the frontend is now able to do many/most of the things that the server is capable of doing:

- Perform validations
- Perform computations
- Perform connections to the backend

So, an intuitive architectural question would be: why does the backend not focus just on rendering initial content while the frontend in the rendered content manages everything else? The answer lies in privacy and security. The code that runs on the client side is visible to the user. In many use cases, we don't want to expose the business logic, the details of the backend components and their inter-relations, and the data involved in the application. So, while the frontend is now capable of doing many things that the back end is capable of, and while offloading responsibility to clients where the content serving is occurring is architecturally valid and reasonable, commercial websites draw a fine boundary between the front and backend based on the privacy and security considerations.

WebWorker

WebWorker implements a threading abstraction. Using WebWorkers, we can create background threads that can theoretically run any script. Page rendition and user interaction that involve massive computation can heavily benefit from worker threads, as they help parallelize the given tasks. Additionally, web workers act as a network proxy, which can asynchronously manage all the network interactions so that the main thread can focus on the frontend interactions. The workers and the main thread do not use normal synchronization primitives to work with shared data; instead, they communicate over non-blocking and asynchronous messaging channel.

The following code snippet shows how to use web workers:

```
1. const w = new Worker('worker.js')
2. w.onmessage = function(event) {
3.   document.getElementById('data').innerHTML = event.data
4. })
```

Question: What is the premise/benefit of communicating between threads over messages as opposed to shared data and synchronization primitives to streamline access?

By components

In this section, we will look at the visual components that make up an HTML page. We illustrate the most common form of each component and provide an example of each, along with how they appear on screen.

Hyperlink

This defines a linkage from the current page to another or from the current section to another. Hyperlink is the most fundamental semantics for web navigation. It practically overarches any other object, making any web object potentially a hyperlink.

The following code uses a hyperlink:

```
1. <html
2. <body>
3. <a href="foo.hml"> click here for foo </a>
4. </html>
```

The following screenshot shows how a hyperlink will be displayed:

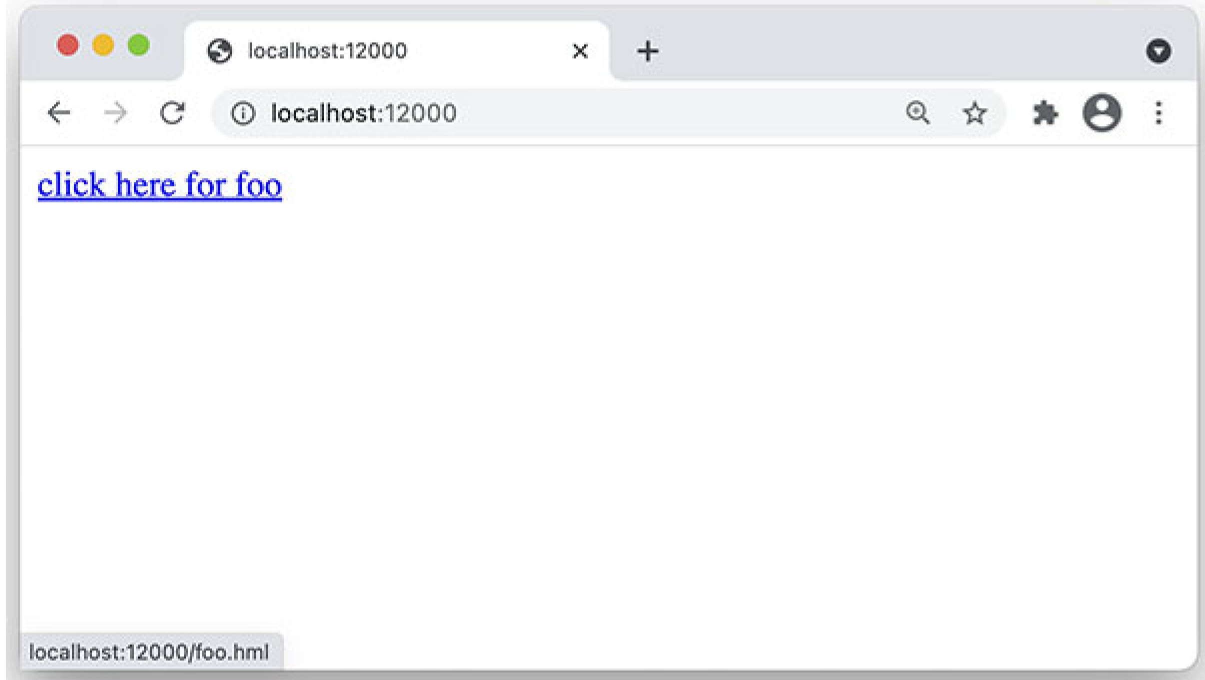

Figure 8.22: *A web page with hyperlink tag*

Article

This defines an independent block of web content. It is a basic construct used to create sections in blogs and journals.

The following code shows the usage of the article tag:

```
1. <html>
2. Welcome to foo's web site!
3. <p>
4. <body>
5. Article:
6. <article>
7. I am a building, in the city of London, very tall, <br>
8. very old, and extremely famous. One of the most <br>
9. important things about me, is its very large domed roof.
   <br>
10. </article>
11. </body>
12. </html>
```

The following screenshot shows how an article tag will be rendered:

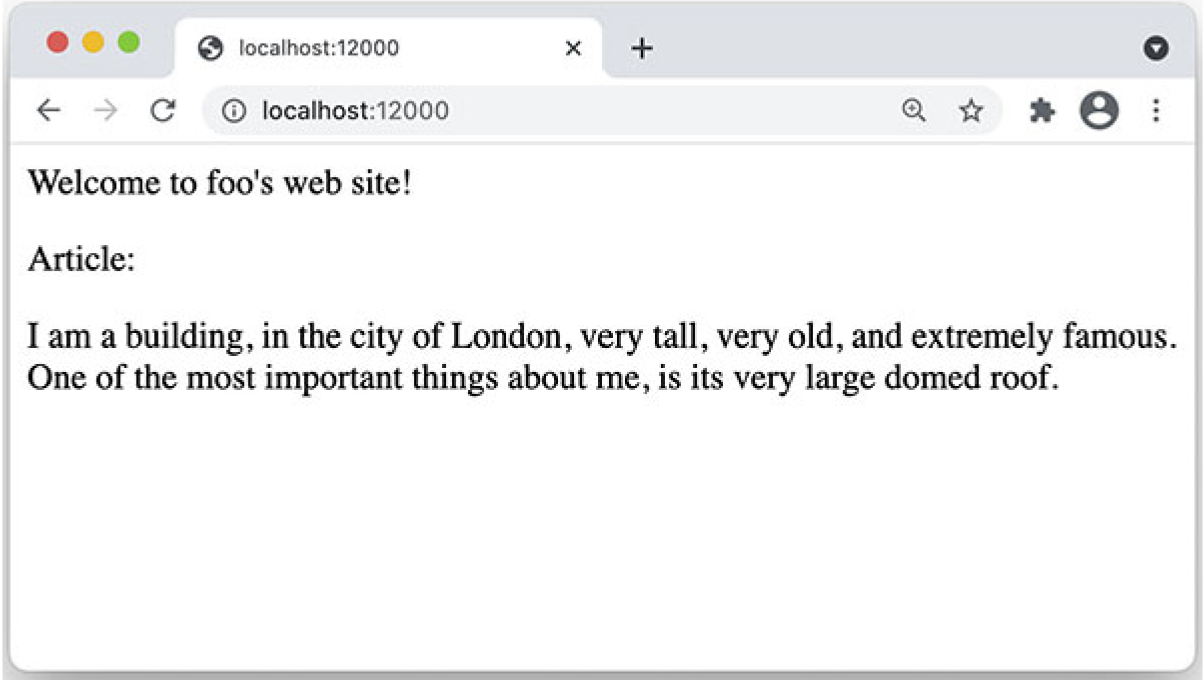

Figure 8.23: *A web page with article tag*

Dialog

This defines a popup frame and is useful for displaying contextual messages and alerts.

The following code shows the usage of the dialog tag:

```
1. <html>
2. Welcome to foo's web site!
3. <p>
4. <body>
5. <dialog open>
6. I am a building, in the city of London, very tall,
7. very old, and extremely famous. One of the most
8. important things about me, is its very large domed roof.
9. </dialog>
10. </html>
```

And the following diagram shows how a dialog tag will be rendered:

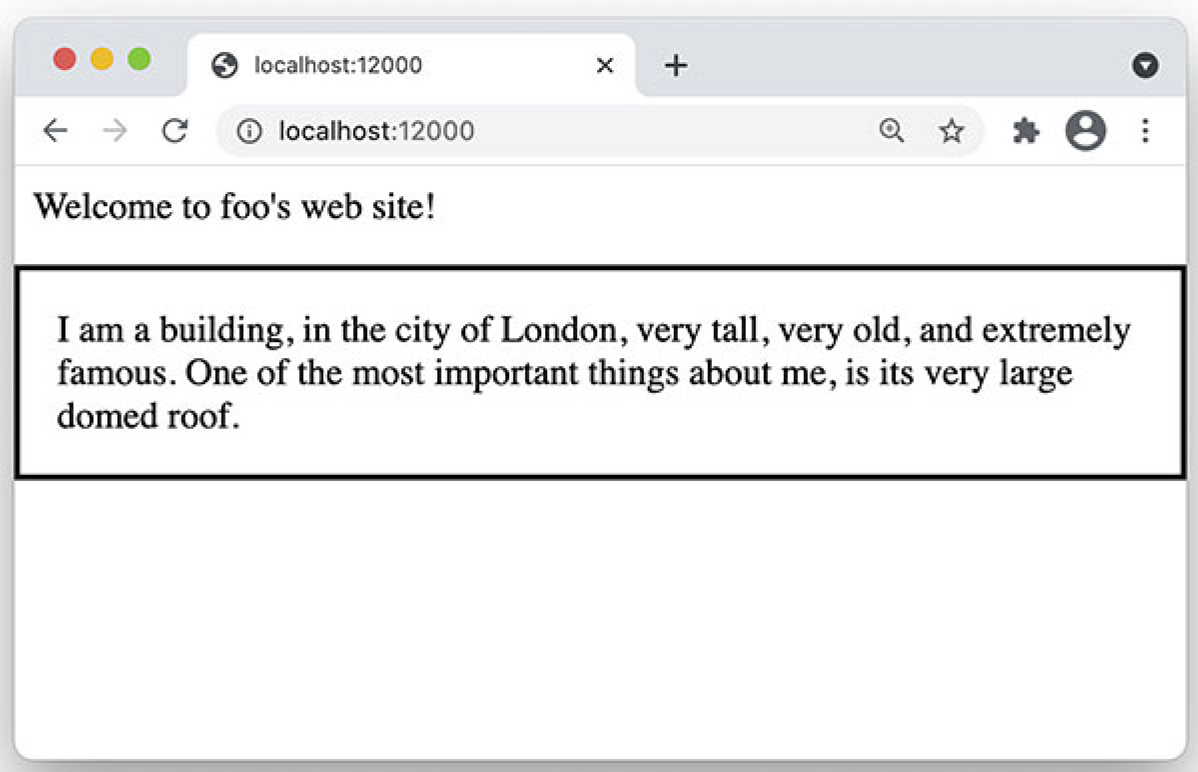

Figure 8.24: *A web page with dialog tag*

Form

A form defines a complex interface for collecting user input and helps the server obtain data from the client. The interface has several fields and attributes to cover various data types and collection methods.

The following code shows the usage of the forms tag:

```
1. <html>
2. Welcome to foo's web site!
3. <p>
4. <body>
5. Registration:    
6. <p>
7. First name:      
8. <input/>
9. <p>
```

```
10. Last name:       
11. <input/>
12. <p>
13. Organization:   
14. <input/>
15. <p>
16. Phone number: 
17. <input/>
18. <p>
19. <button type="button">submit</button>
20. <p>
21. </html>
```

The following screenshot shows the resultant web form:

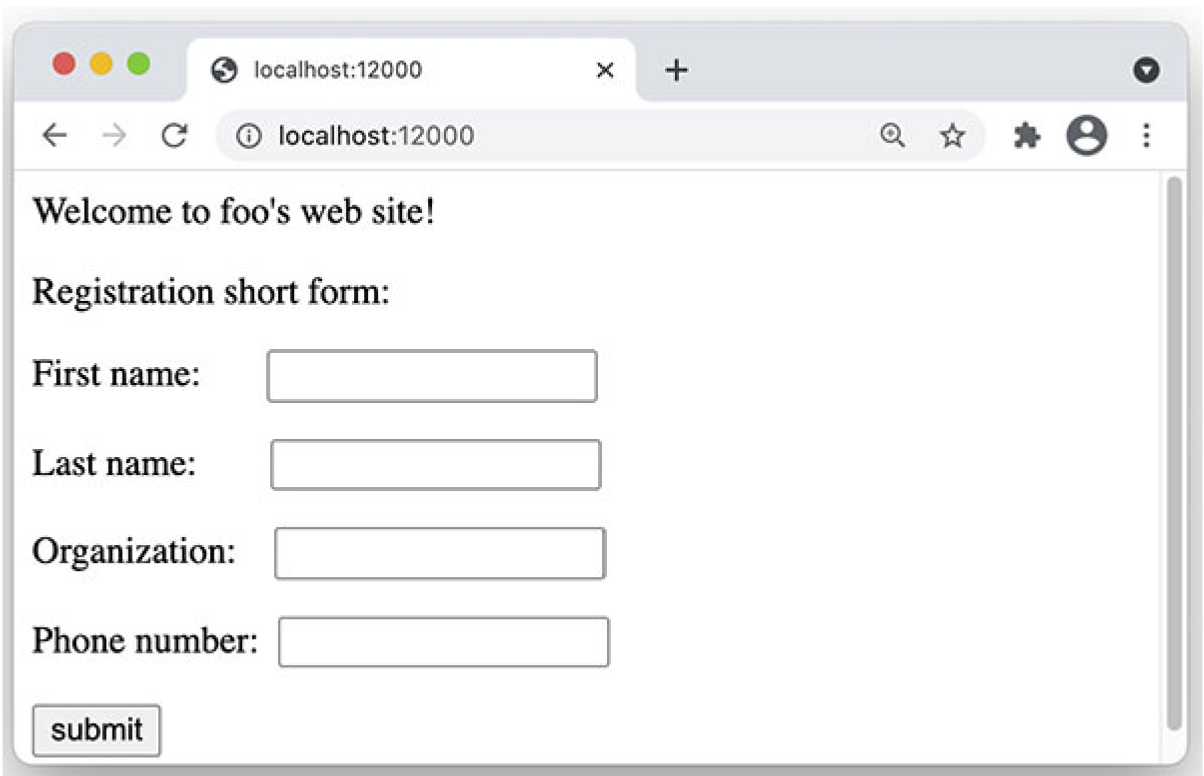

Figure 8.25: *A web page with web form*

Options/dropdown

It defines a list for selecting a choice and is useful when the input set is known but the input itself is not.

The following code shows the usage of the option tag:

```
1. <html>
2. <body>
3. <h3>Choose a fruit:</h3>
4. <select id="fruits">
5.   <option value="apple">Apple</option>
6.   <option value="orange">Orange</option>
7.   <option value="mango">Mango</option>
8.   <option value="strawberry">Strawberry</option>
9. </select>
10. <button type="button">Add to cart</button>
11. </body>
12. </html>
```

And the following screenshot shows how it will look in the browser:

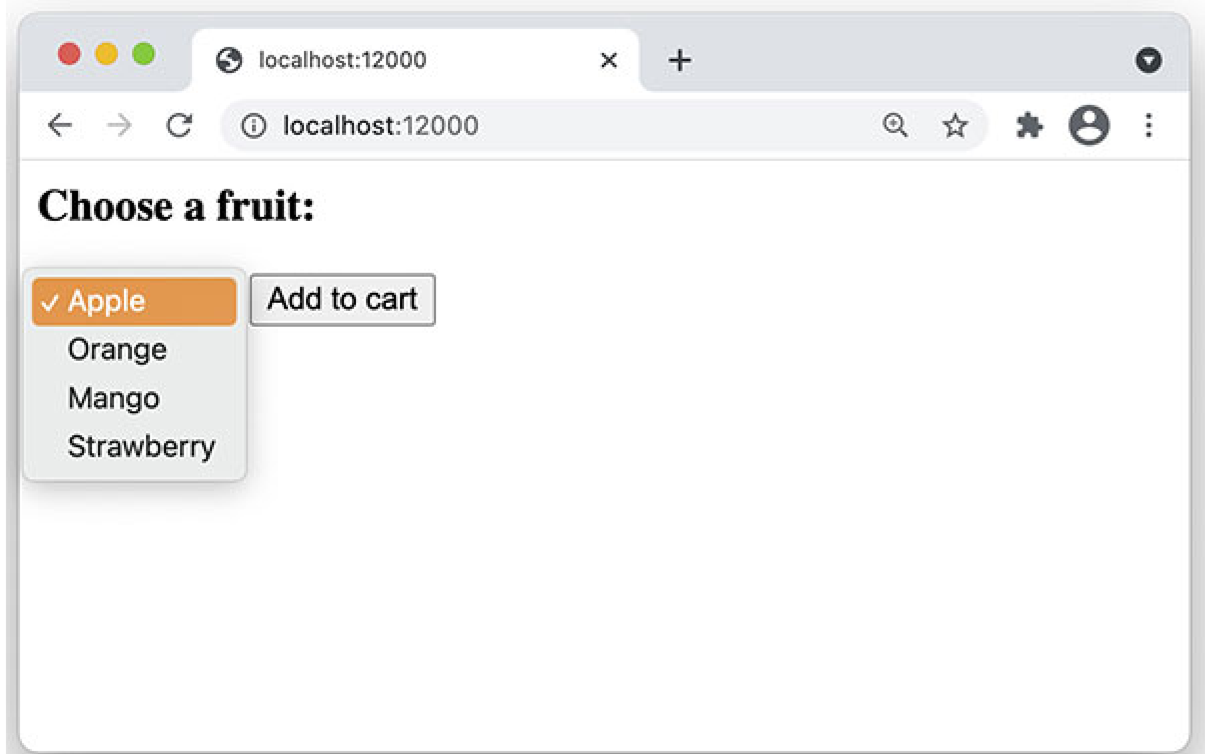

Figure 8.26: A web page with option tag

Table

This defines a tabular structure, and the attributes define rows, columns, and properties relating to their physical appearance.

The following code shows the usage of the table, **`th`** (table header), **`tr`** (table row), and **`td`** (table data) tags:

```
1. <html>
2. <head>
3. <body>
4. <style>
5. table, th, td {
6.   border: 1px solid black;
7. }
8. </style>
9. Table for foo
10. <p>
11. <table>
12.   <tr> <th>Fruit</th> <th>Cost</th> <th>Discount</th> </tr>
13.   <tr> <td>Apple</td> <td>$4</td> <td>25%</td> </tr>
14.   <tr> <td>Orange</td> <td>$5</td> <td>30%</td> </tr>
15.   <tr> <td>Mango</td> <td>$7</td> <td>10%</td> </tr>
16. </table>
17. </body>
18. </html>
```

And the following diagram shows the resultant table:

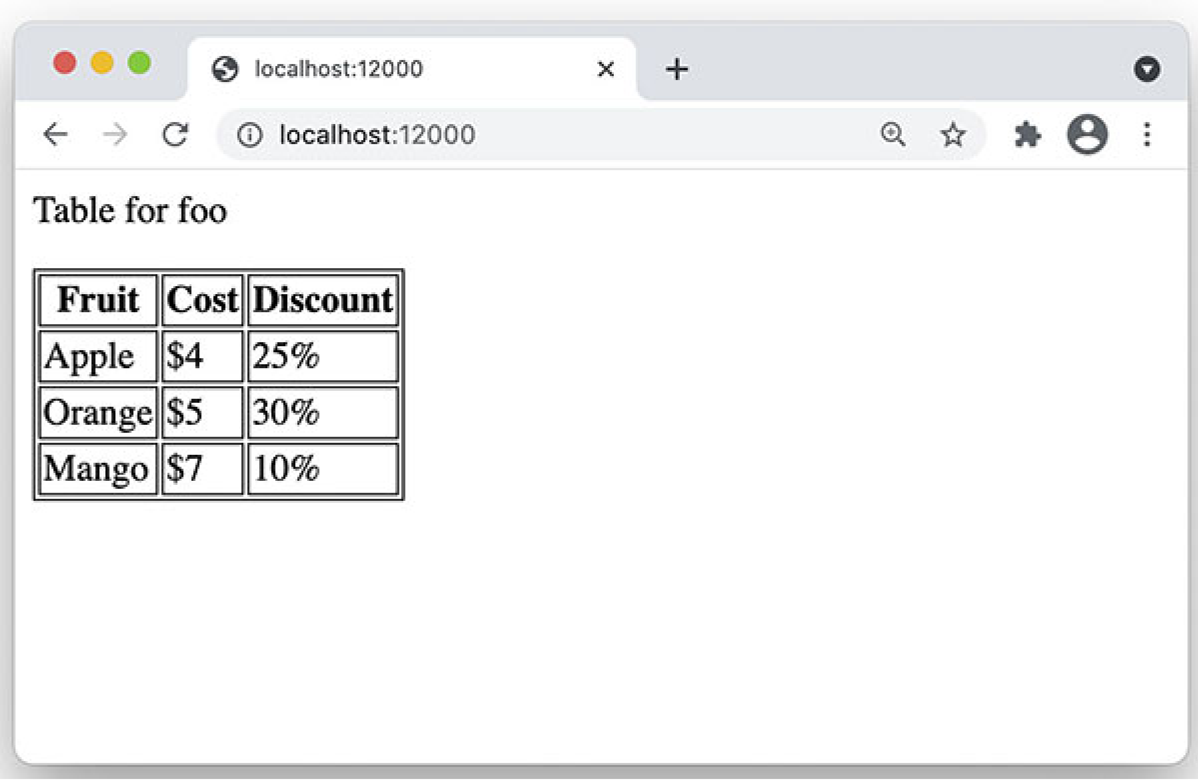

Figure 8.27: *A web page with tabular data*

Image

This defines an image reference and is useful to embed pictures on a web page.

The following code shows the usage of the image tag:

```
1. <html>
2. <body>
3. Welcome to foo's site!
4. <p>
5. <img src="/img.png" width="150" height="100">
6. </body>
7. </html>
```

The following screenshot diagram shows the resultant image:

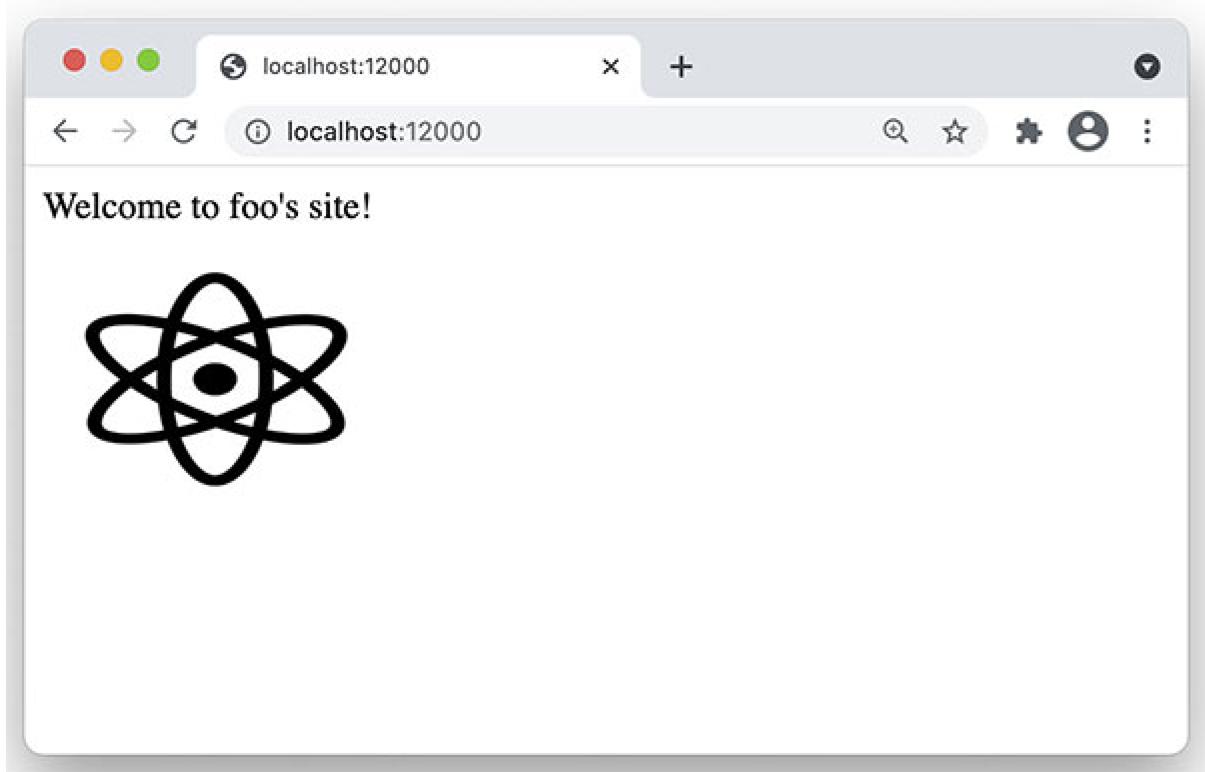

Figure 8.28: *A web page with image*

Audio

This defines an audio reference and is useful to embed audio clips in a web page. The following code shows the usage of the audio tag:

```
1. <html>
2. <body>
3. Welcome to foo's web site!
4. <p>
5. <audio controls>
6. <source src="foo.mp3" type="audio/mpeg">
7. </audio>
8. </body>
9. </html>
```

And the following screenshot shows a web page that embeds an audio control:

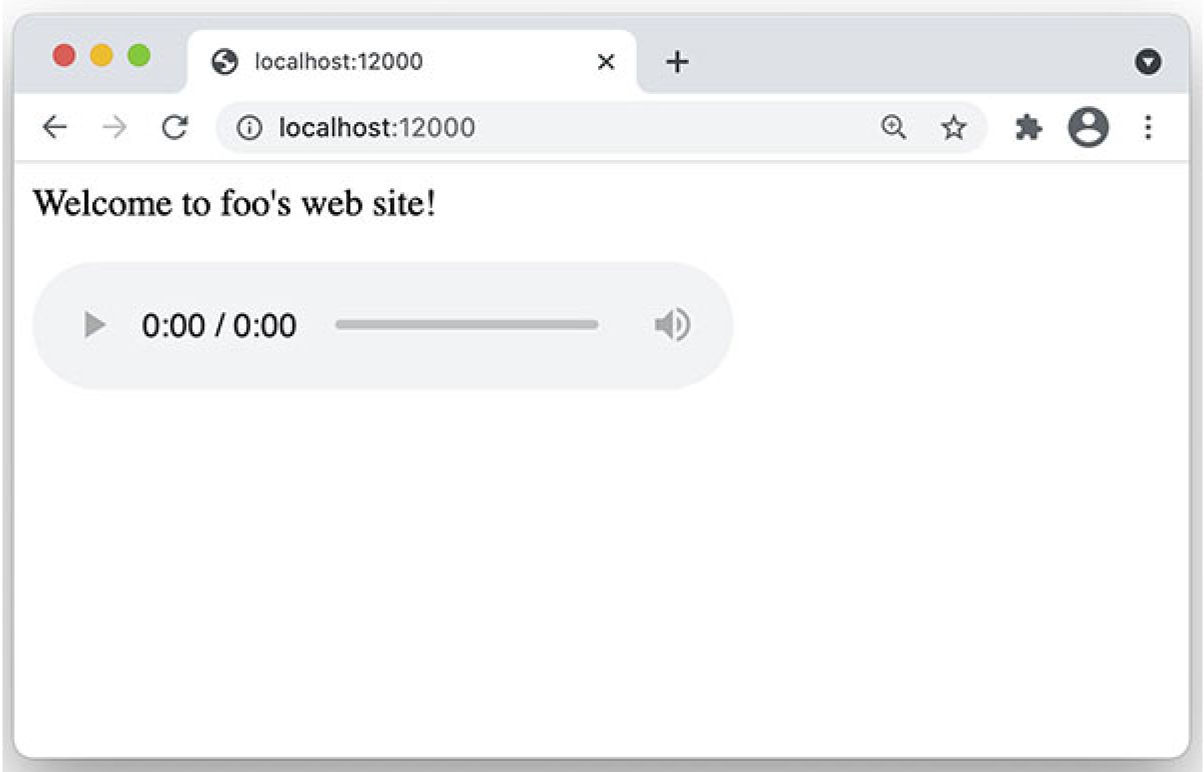

Figure 8.29: *A web page with audio control*

Video

This defines a video reference and is useful to embed video clips in a web page. The following code shows the usage of the video tag:

```
1. <html>
2. <body>
3. Welcome to foo's website!
4. <p>
5. <video controls>
6. <source  src="foo.mp4" type="video/mp4">
7. </video>
8. </body>
9. </html>
```

And the following screenshot shows how it gets rendered:

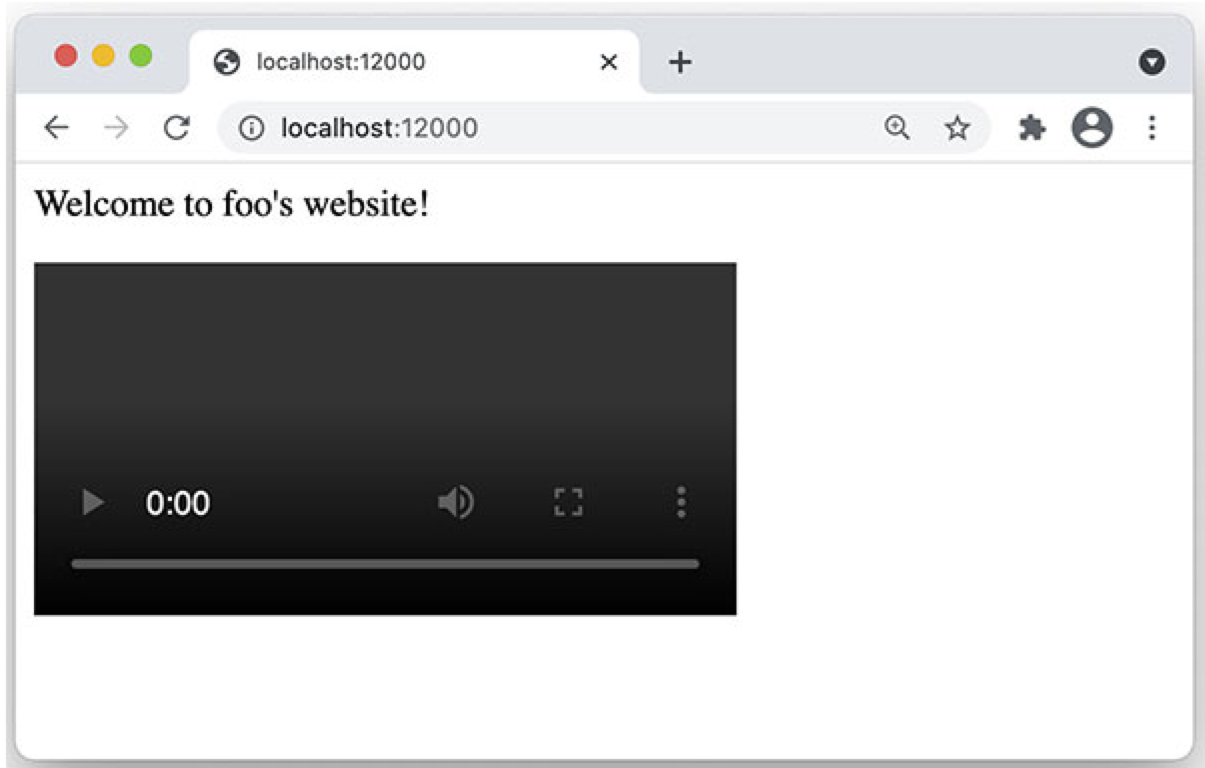

Figure 8.30: *A web page with video control*

Script

This defines a program in another language to be embedded that gets executed instead of rendition. So, the resultant action of the script is considered to be the result of the rendition of the element. At present, only JavaScript is supported as a target script. The following code shows the usage of JavaScript embedding in HTML using the script tag:

```
1. <html>
2. Welcome to foo's web site!
3. <p>
4. <body>
5. <script>
6. alert('hello from script!')
7. </script>
8. </html>
```

And the following screenshot shows the resultant action in the web page:

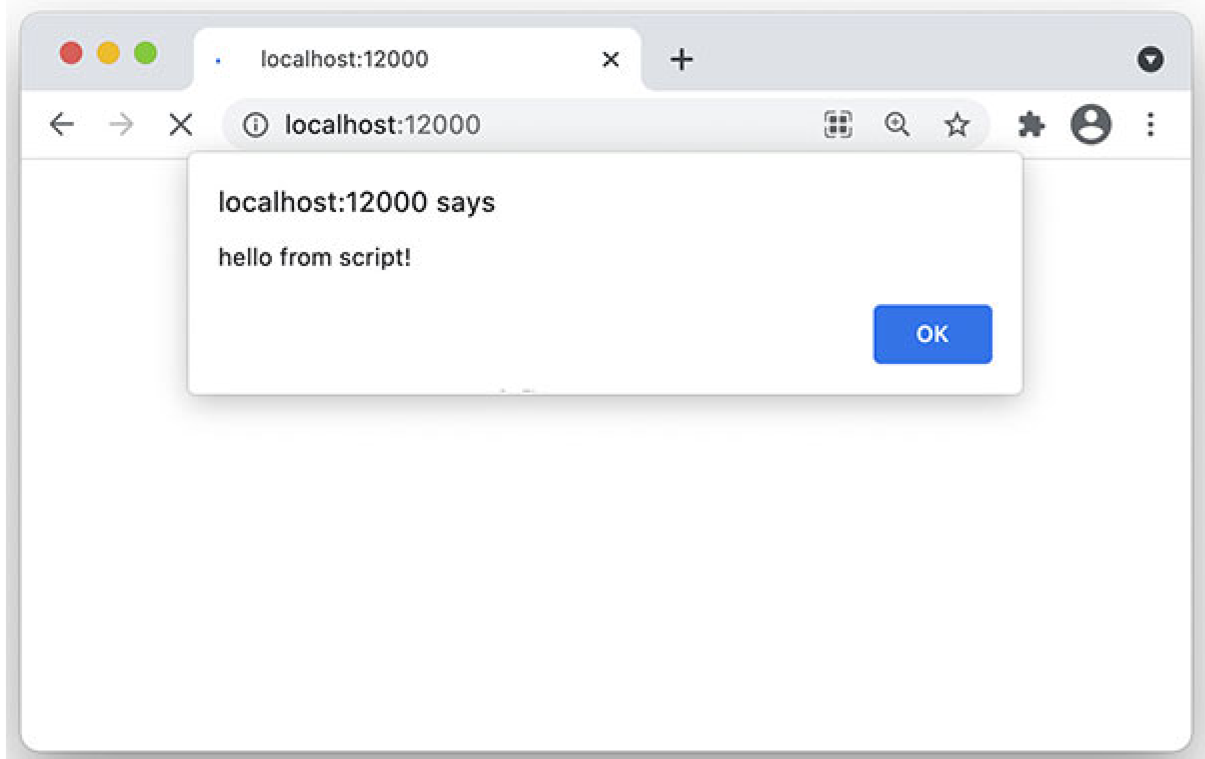

Figure 8.31: *A web page with JavaScript in action*

Website: Advanced features

In this section, we will look at some of the advanced features that make our website attractive and versatile. These features are evolved as a natural extension to the web page design and integrated into the web standards, or they are contributions from third-party vendors. These features enhance the usability of web pages two-fold and contribute to improved user experience.

Google APIs

So far, we have seen examples of common constructs and interfaces that are useful on a web page. Extending the use cases further, how do we add more complex objects into a web page? Complexity should be in terms of their appearance, features, data model and behavior, and response to user interaction.

As we know with moderately complex components like a table, the HTML code grows in terms of complexity. This complexity leads to the introduction of errors. We have seen examples of a manually coded HTML

table missing a column, data element skewed a little, and so on. It is natural that the readability and manageability of code is inversely proportional to the complexity.

What would be the complexity of an office tool, such as a spreadsheet? A spreadsheet is a superset of a table, with several extended features, such as a collection of mathematical functions, and higher order visualization of data, such as graphs and charts. Obviously, the page becomes unmanageable. Think of the server responsible for composing such a page and dispatching it to the client, along with other business logic processing and their responses!

Google APIs for office documents come from an architectural perspective of HTML as a service. What if a server dedicates the business logic pertinent to one complex object rendered in our page, and the server handles all the interactions, like events, event handling, re-rendering of data, and computations? Google APIs are exactly that. They provide an interface to the client for embedding complex but isolated objects into the web pages that are self-sustainable. The following code shows the usage of object embedding with the iframe tag:

```
1. <html>
2. <body>
3. Welcome to foo's web site!
4. <p>
5. <iframe align="center" src="https://docs.google.com/document/d/
6. 1t5CrKW4jIBUcn4KZXZRJVYL4OdZigkj-miVGY31P36I/edit?
7. usp=sharing" height="200" width="400"></iframe>
8. </body>
9. </html>
```

And the following diagram shows how a Google document is embedded into a web page:

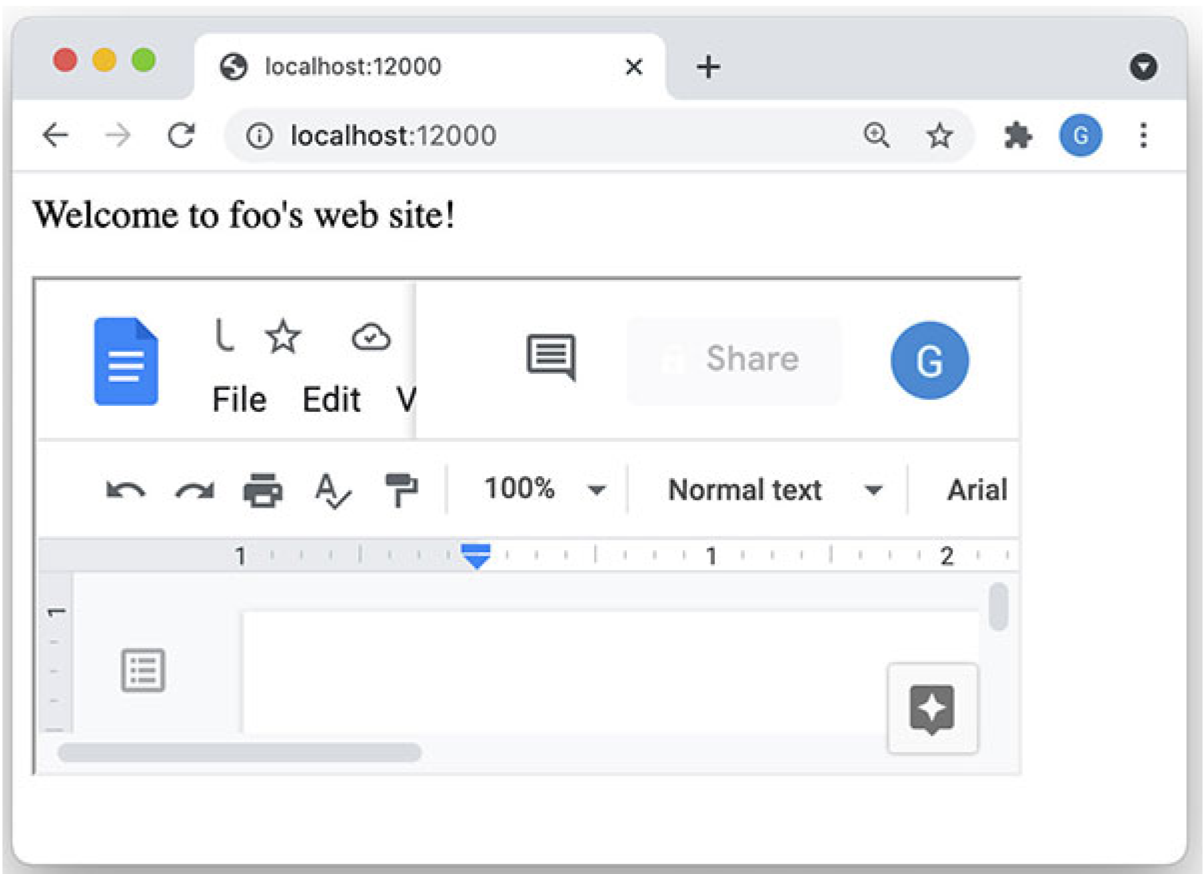

Figure 8.32: *A web page with embedded Google document*

Assignment: Develop a small web page and transform it such that the entire content can be embedded into another page by removing only the over-arching ‘<html>’ and ‘</html>’ tags. Observe how the new page (embedded page) behaves and isolates itself from the master page (embedding page) in terms of styles, attributes, and behavior.

Similarly, the following code embeds a Google spreadsheet into a page:

```
1. <html>
2. <body>
3. Welcome to foo's web site!
4. <p>
5. <iframe
   src="https://docs.google.com/spreadsheets/d/1roC4JOoW1Y59Py
   n-iXVM_hyFmojUXI3et8XxakEy0ZY/edit#gid=0"      height="200"
   width="400"></iframe>
```

```
6. </body>
7. </html>
```

And the resultant view is as follows:

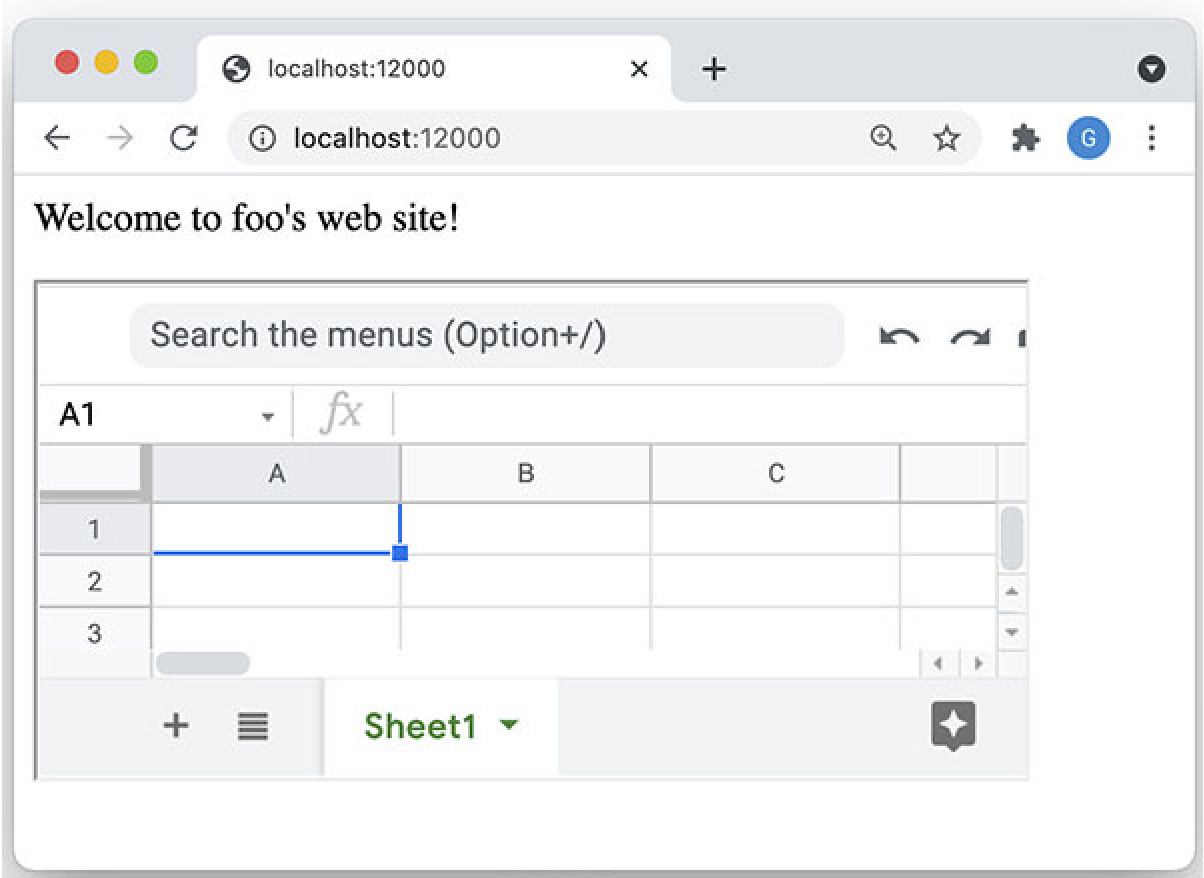

Figure 8.33: *A web page with embedded Google spreadsheet*

Similarly, the following code embeds a Google presentation into a web page:

```
1. <html>
2. <body>
3. Welcome to foo's web site!
4. <p>
5. <iframe
   src="https://docs.google.com/presentation/d/1VHW_itra4gq9Th
   jTGPAQrm0fuRWjEiHZaFz7niaggMs/edit#slide=id.p" height="200"
   width="400"></iframe>
```

```
6. </body>
7. </html>
```

And the following screenshot shows the resulting Google presentation:

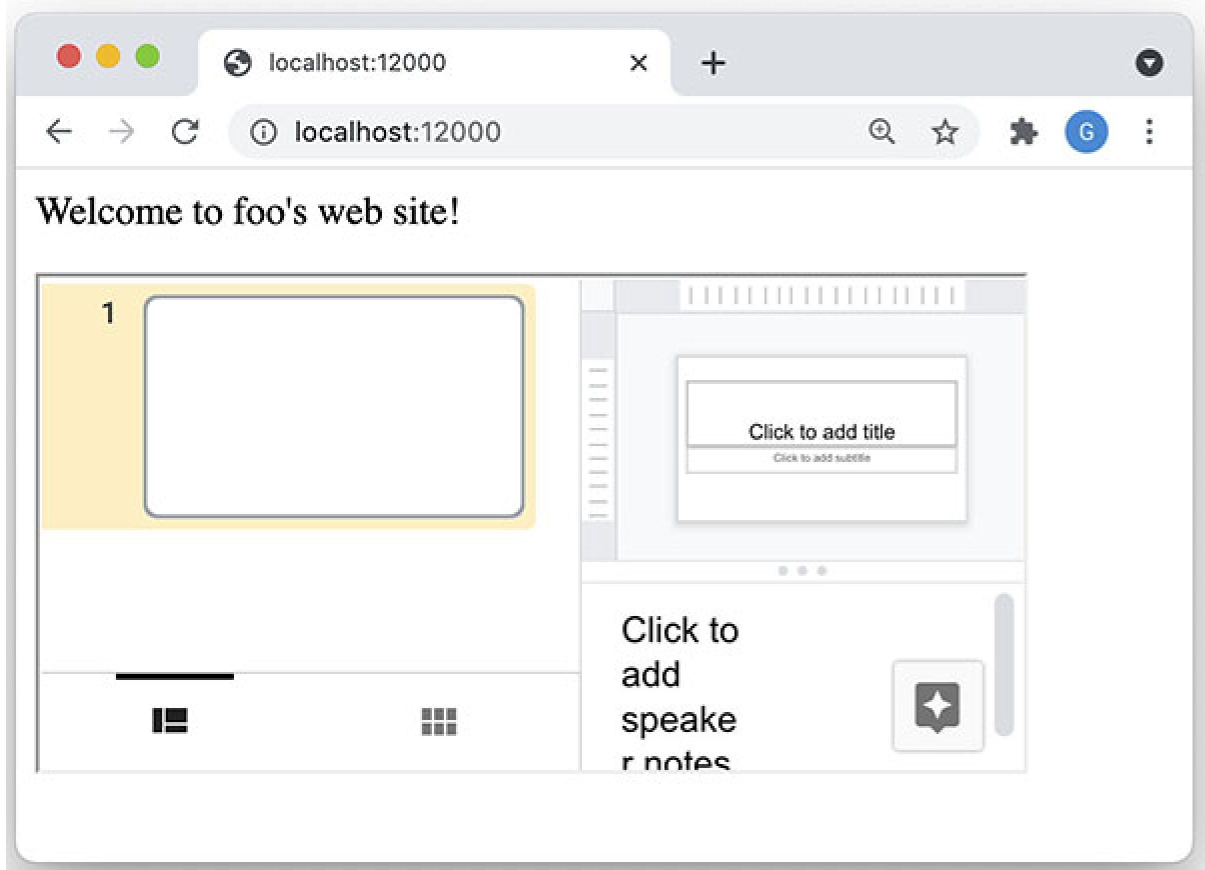

Figure 8.34: *A web page with embedded Google presentation*

The following code shows how to embed a custom Google chart into a web page:

```
1. <html>
2. <head>
3. <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>
4. <script type="text/javascript">
5. google.charts.load('current', {'packages':['corechart']});
6. google.charts.setOnLoadCallback(draw);
7. function draw() {
```

```
8.  var data = new google.visualization.DataTable();
9.  data.addColumn('string', 'Fruit');
10. data.addColumn('number', 'percentage');
11. const rows = []
12. rows.push(['Apple', 30])
13. rows.push(['Orange', 40])
14. rows.push(['Mango', 30])
15. data.addRows(rows)
16. const opt = {'width':600, 'height':400}
17. const chart = new
    google.visualization.PieChart(document.getElementById('frui
    t'))
18. chart.draw(data, opt)
19. }
20. </script>
21. </head>
22. <body>
23. <div id="fruit"></div>
24. </body>
25. </html>
```

And the following screenshot shows the custom chart rendered in the web page:

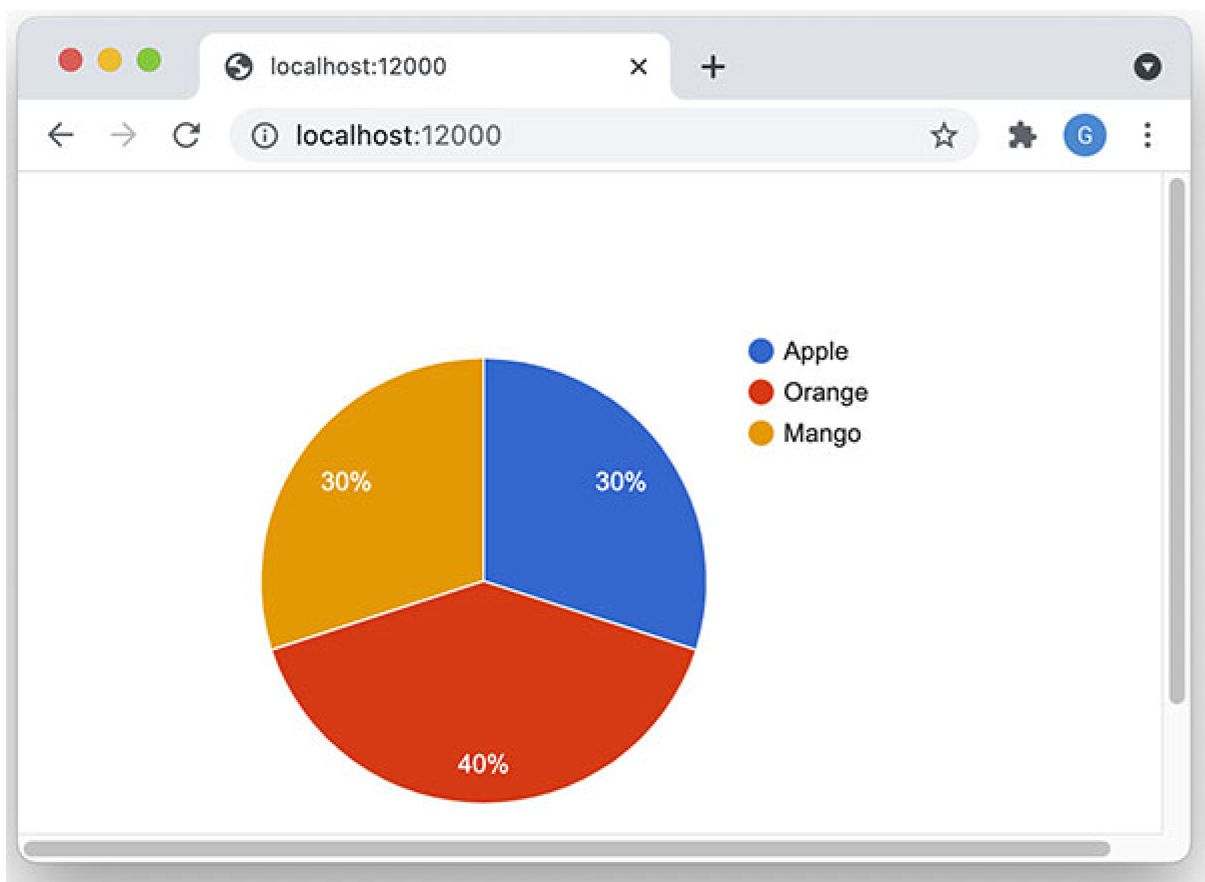

Figure 8.35: *A web page with embedded Google charts*

Embedding: Maps and social media

This is a similar use case as explained earlier but semantically slightly different in terms of object embedding. In this style, the map is expressed in a division of the page using the "**<div>**" HTML tag. Google Maps provide powerful abstraction for navigating in the real world.

The following code shows the embedding of Google maps into a web page:

```
<html>
<head>
<body>
<iframe src="https://www.google.com/maps/embed?pb=!1m10!1m8!1m
3!1d248849.84916296526!2d77.6309395!3d12.9539974!3m2!1i1024!2i7
68!4f13.1!5e0!3m2!1sen!2sin!4v1617636691038!5m2!1sen!2sin"
width="600" height="450" style="border:0;" allowfullscreen=""
loading="lazy"></iframe>
```

```
1. </body>
2. </html>
```

And the following screenshot depicts the resulting Google Map:

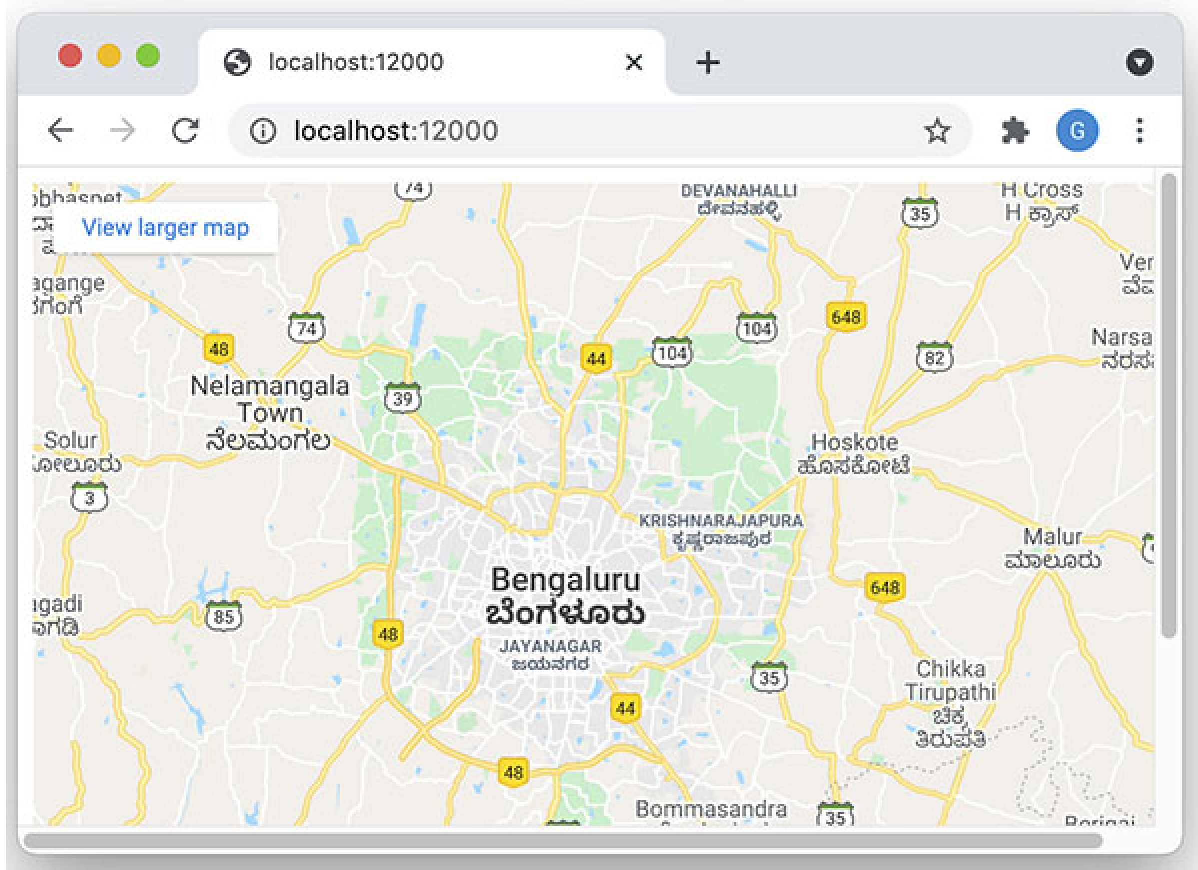

Figure 8.36: *A web page with embedded Google Maps*

Similarly, social media share icons provide ways to share your contextual content with your contacts, which significantly improves communication. An alternative is to copy the link of the web page, open your social media application, and share the link. The drawbacks include the additional effort required for copy pasting and that the entire page becomes the subject for sharing as opposed to any customizations.

The following code shows how to embed the social media share feature in your web page:

```
1. <html>
2. Welcome to foo's web site!
3. <p>
```

```
4. <body>
5. <a class="twitter-share-button"
6.                     href="https://twitter.com/intent/tweet?
   text=I%20am%20a%20building,%20in%20the%20city%20of%20London
   ,%20very%20tall,%20very%20old,%20and%20extremely%20famous.%
   20One%20of%20the%20most%20important%20things%20about%20me,%
   20is%20its%20very%20large%20domed%20roof.">
7. Tweet</a>
8. <p>
9. Unregistered view:
10. <article>
11. I am a building, in the city of London, very tall, very
    old, and extremely famous. One of the most important things
    about me, is its very large domed roof.
12. </article>
13. </html>
```

And following screenshot shows how the share feature is embedded in the page:

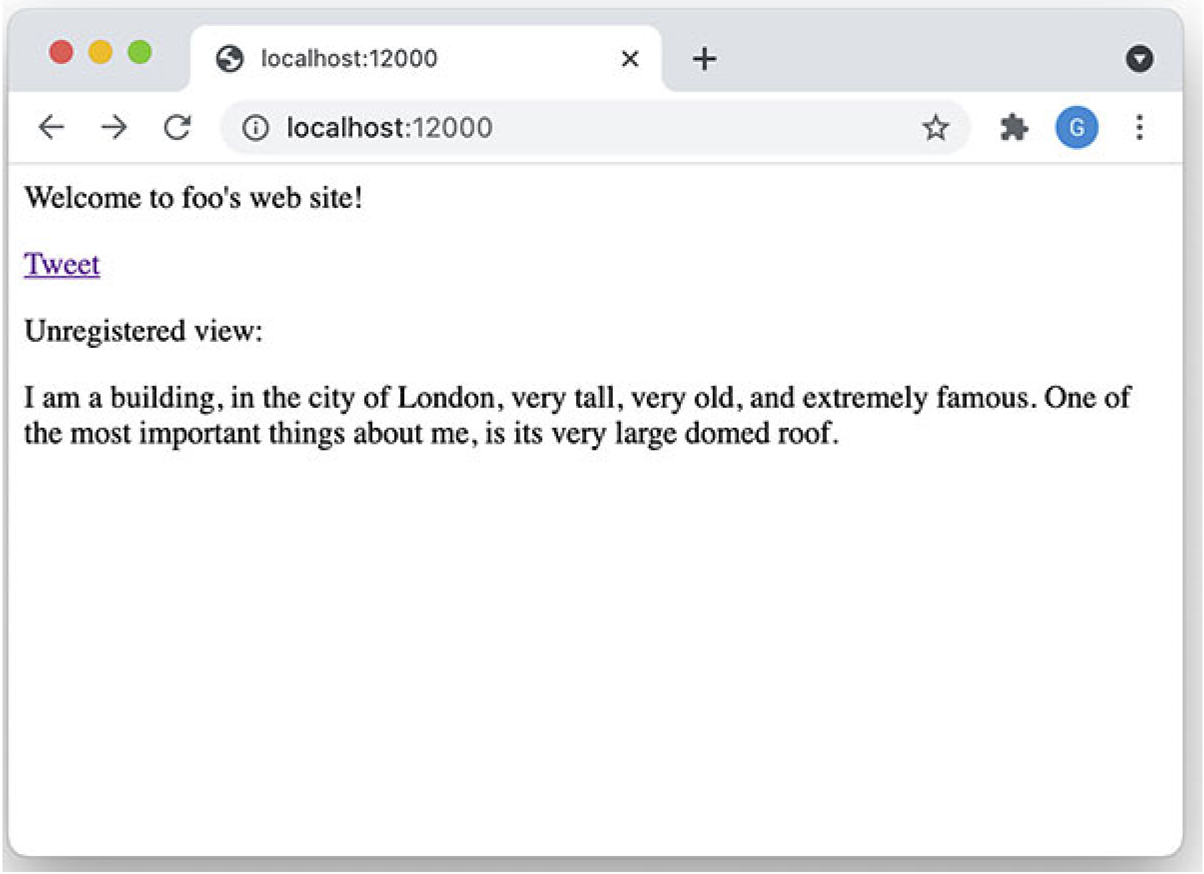

Figure 8.37: *A web page with embedded social media share button*

And the following screenshot shows the target page of the share link:

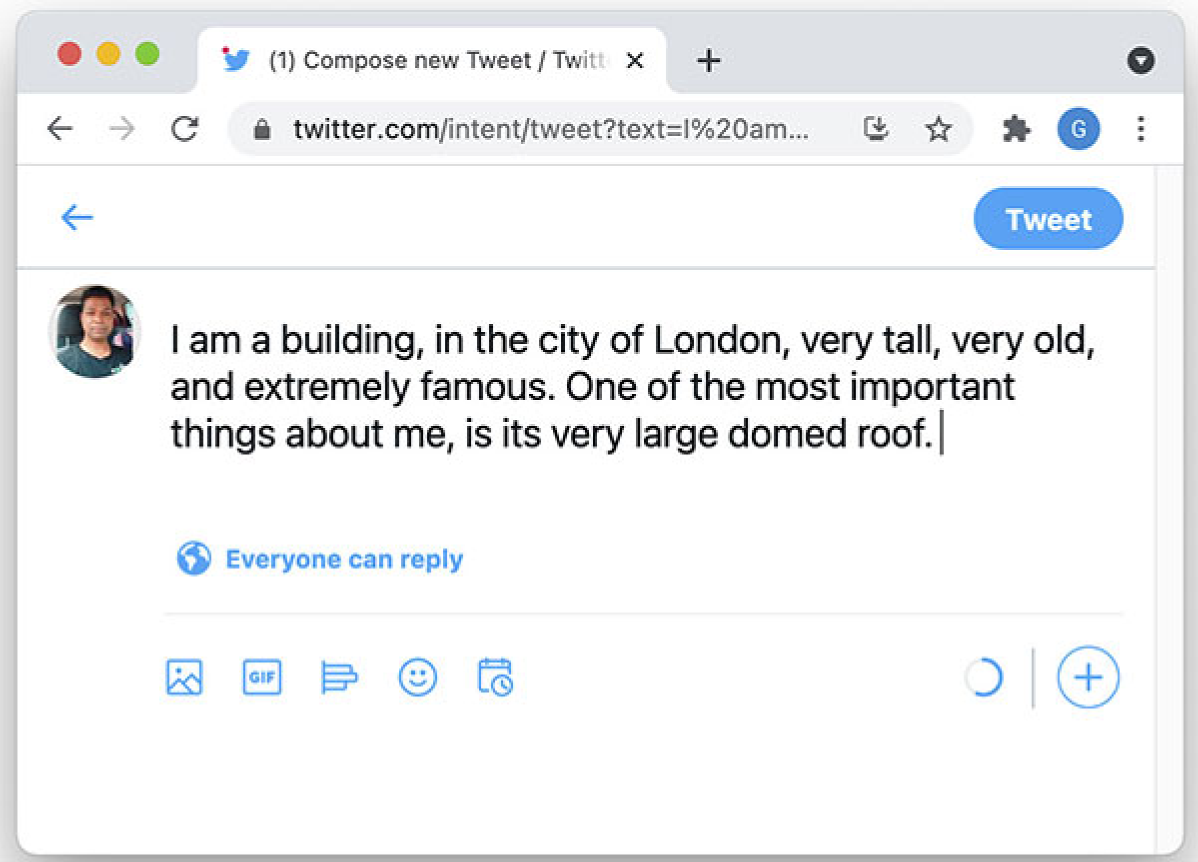

Figure 8.38: *A web page with embedded social media*

Pagination

When the backend responds with data that the frontend renders in the page, what happens if there's more data than the page can hold? As we discussed in the user experience section, dumping a lot of content leads the page to have a scroll bar and exhibits poor user experience.

A known technique to solve this is to conveniently cut the data at logical boundaries and amounts that can roughly fit in a page and provide navigational controls for the rest of the data. This is called **pagination**.

Pagination has an added advantage. It can focus on delivering the first page to the client with utmost responsiveness and offload the processing of the rest of the page at its own sweet pace, depending on how the backend is designed.

The following code shows the implementation of simple pagination:

```
1. <html>
```

```
2. <head>
3. <style>
4. .pagination a {
5.   color: red; float: left;
6.   padding: 10px 16px;
7.   text-decoration: none;
8. }
9. </style>
10. </head>
11. Welcome to foo's web site!
12. <p>
13. Page 1:
14. <br>
15. <article>
16. I am a building, in the city of London, very tall, <br>
17. very old, and extremely famous. One of the most <br>
18. important things about me, is its very large domed roof.
    <br>
19. </article>
20. <body>
21. <div class="pagination">
22.   <a href="pagination1.html">&laquo;</a>
23.   <a href="pagination1.html">1</a>
24.   <a href="pagination2.html">2</a>
25.   <a href="pagination3.html">3</a>
26.   <a href="pagination3.html">&raquo;</a>
27. </div>
28. </html>
```

And the following screenshot shows the resulting paginated web content:

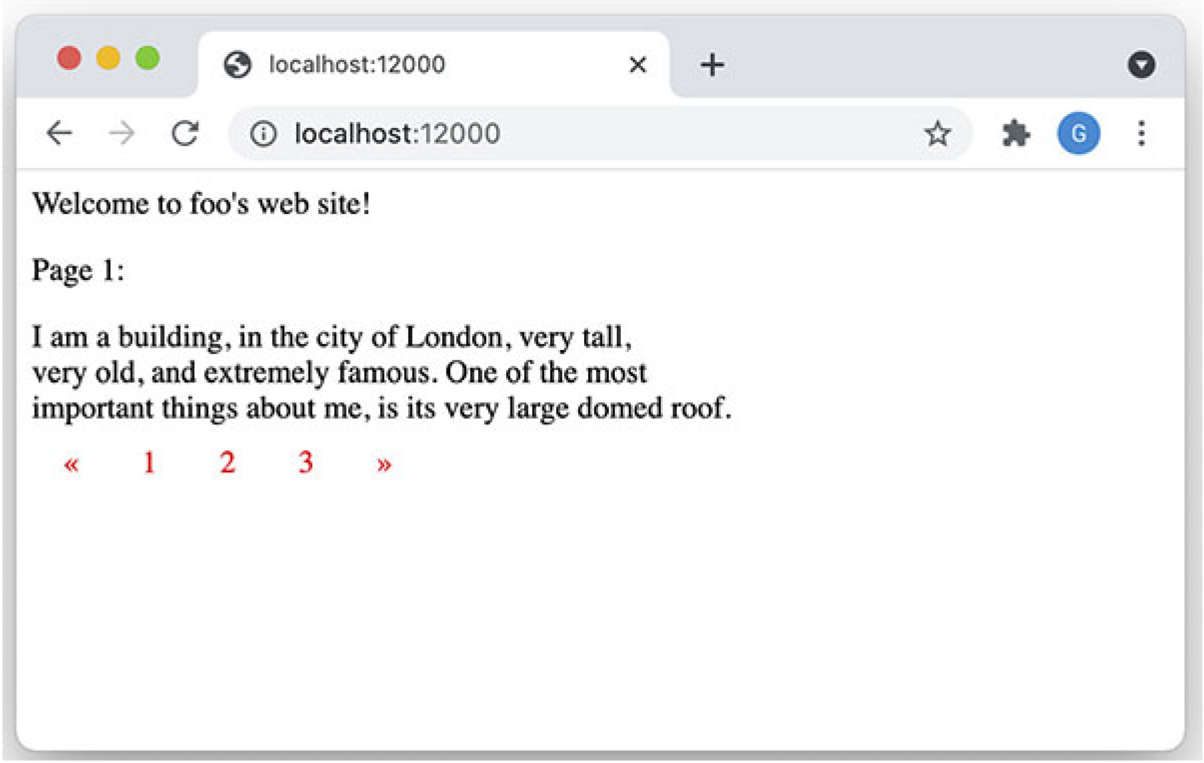

Figure 8.39: *A web page with paginated content*

Search

This refers to embedding a trivial search engine or a custom search feature; both are equally useful in many websites. While a trivial search engine provides you with the option of searching the Internet while still remaining on the website, the custom search feature performs the search within the website.

Creating a search form is very easy, but what makes it complex is how the query is processed at the server side.

The following code shows how to embed a search feature in our web page:

```
1. <html>
2. <body>
3. <form action="https://www.google.com/search"
   target="_self">
4. <input name="q" type="text">
5. <button type="submit">Search</button>
```

6. </form>
7. </body>
8. </html>

The following screenshot shows how the search feature is embedded in the page:

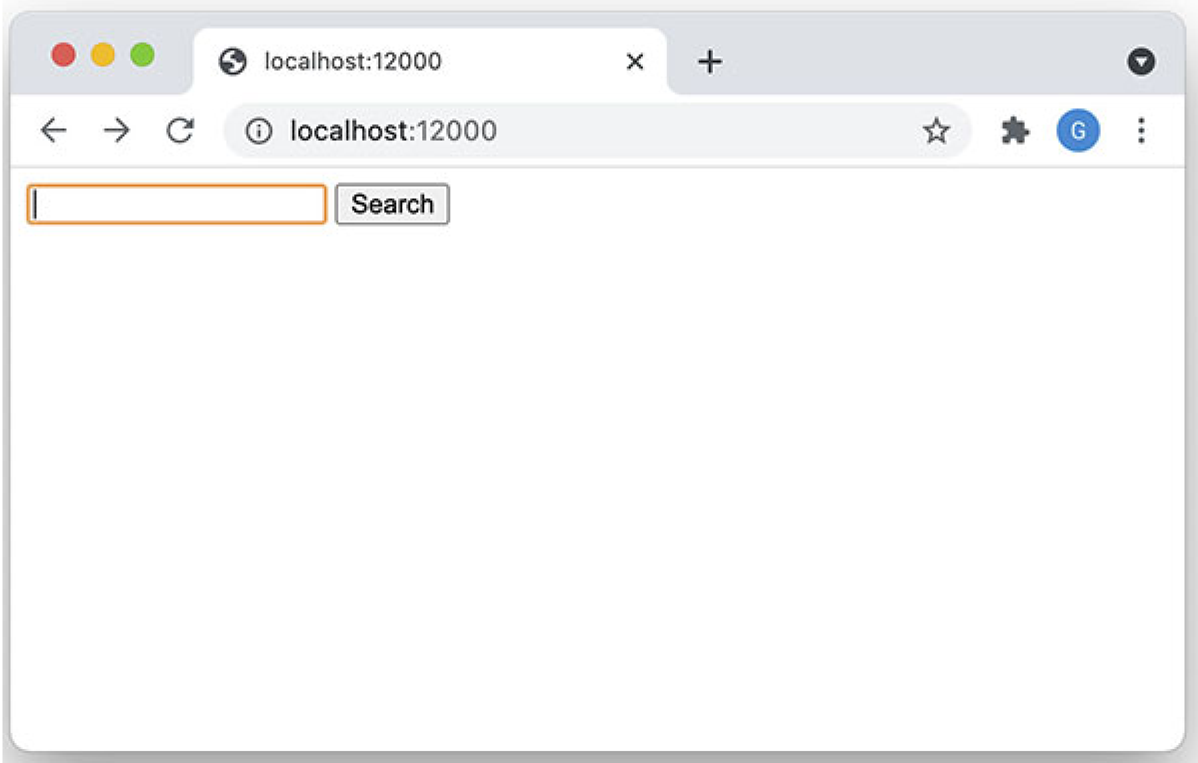

Figure 8.40: *A web page with search option*

The next screenshot shows the resultant page of a search performed:

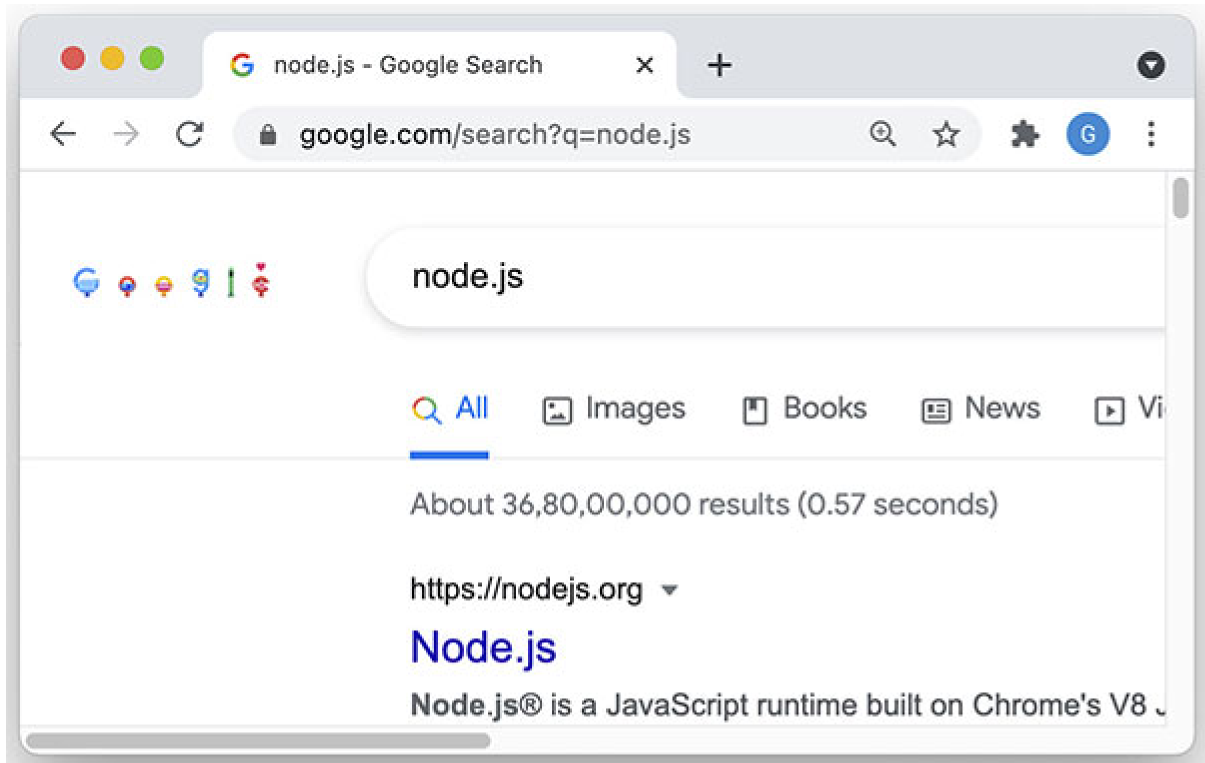

Figure 8.41: *A web page with search result*

Filters

Filters are predicates that control the search behavior. The most common filter types are: filter by date range and by match type. Similarly, sorters are functions applied on result ordering. The most common sort types are: sort by date and by relevance.

The following code shows how to apply filter controls:

```
1. <html>
2. <body>
3. <form method="POST">
4. <select name="range" id="range">
5.   <option value="day">this day</option>
6.   <option value="week">this week</option>
7.   <option value="month">this month</option>
8.   <option value="year">this year</option>
9. </select>
```

```
10. <input name="q" type="text">
11. <button type="submit">Search</button>
12. </body>
13. </html>
```

The following screenshot shows how the filter control is rendered in the page:

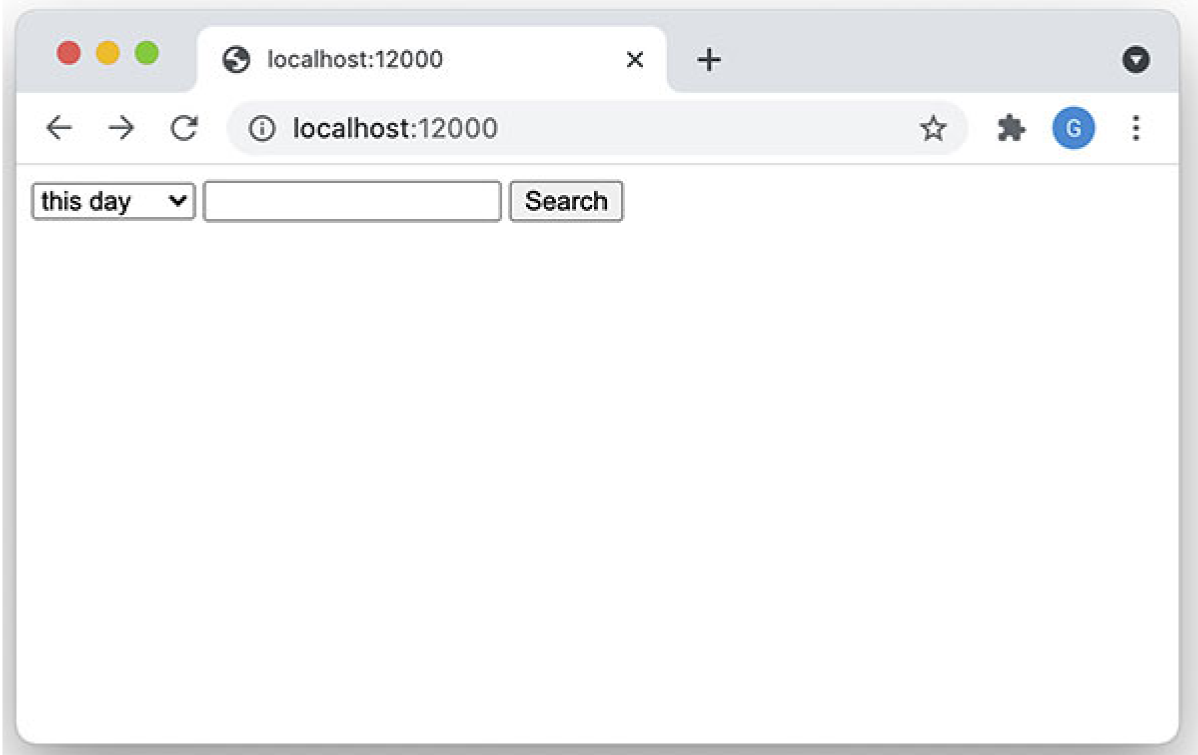

Figure 8.42: *A web page with filter controls*

It is up to the backend to decide what to do with the filter options. The following screenshot shows a web page that simply printed back the filter options to show how it can be retrieved:

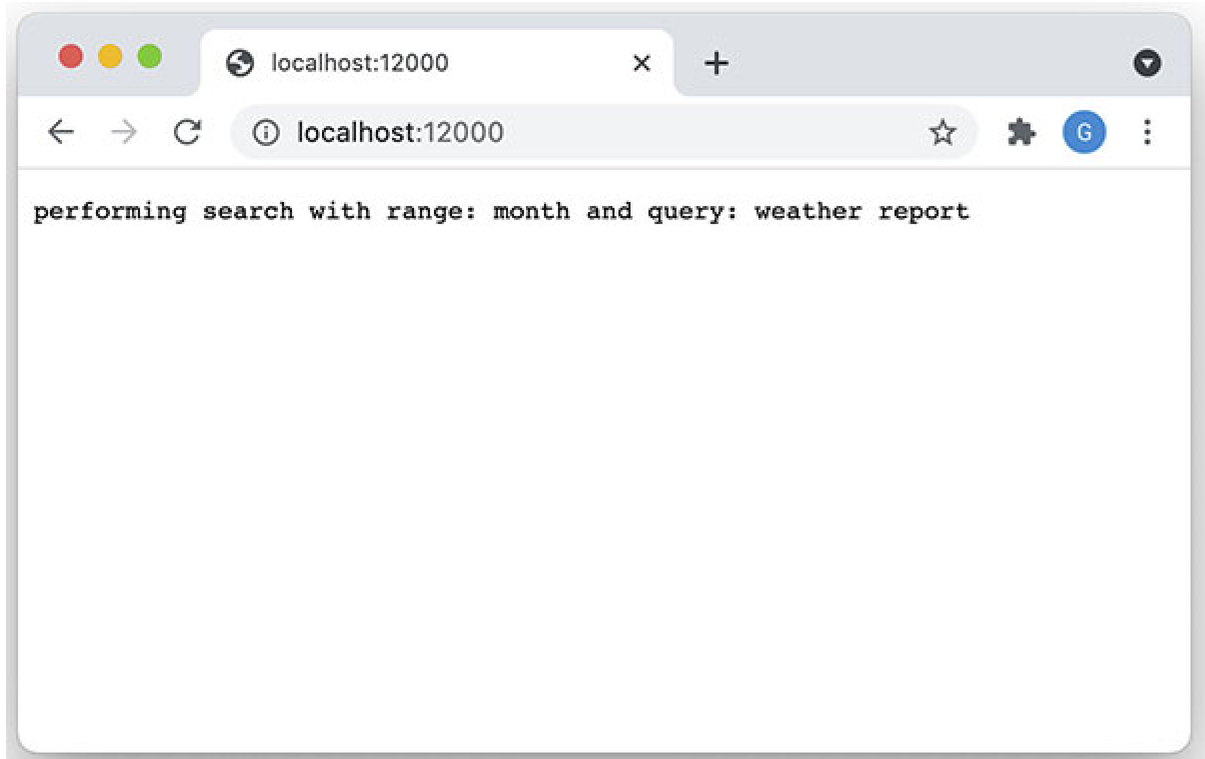

Figure 8.43: *A web page with filtered content*

Geolocation indexing

In many use cases, the server would need to know your current location. A simple example is a home delivery application wanting to know your precise location rather than address; this can then be mapped to know the street address using Google Maps integration.

The following code shows the usage of geolocation indexing:

```
1. <html>
2. <head>
3. <style>
4. table, th, td {border: 1px solid black;}
5. </style>
6. </head>
7. <body>
8. Welcome to foo's web site!
```

```
9. <p>
10. Your location details:
11. <table>
12.   <tr>
13.     <th>Longitude</th>
14.     <th>Latitude</th>
15.   </tr>
16.   <tr>
17.     <td id="long">0</td>
18.     <td id="lati">0</td>
19.   </tr>
20. </table>
21. <script>
22. var long = document.getElementById("long")
23. var lati = document.getElementById("lati")
24.   navigator.geolocation.getCurrentPosition((pos) => {
25.   lati.innerHTML = Math.round(pos.coords.latitude * 100) / 100
26.   long.innerHTML = Math.round(pos.coords.longitude * 100) / 100
27. })
28. </script>
29. </body>
30. </html>
```

And the following screenshot shows how the location is displayed on the page:

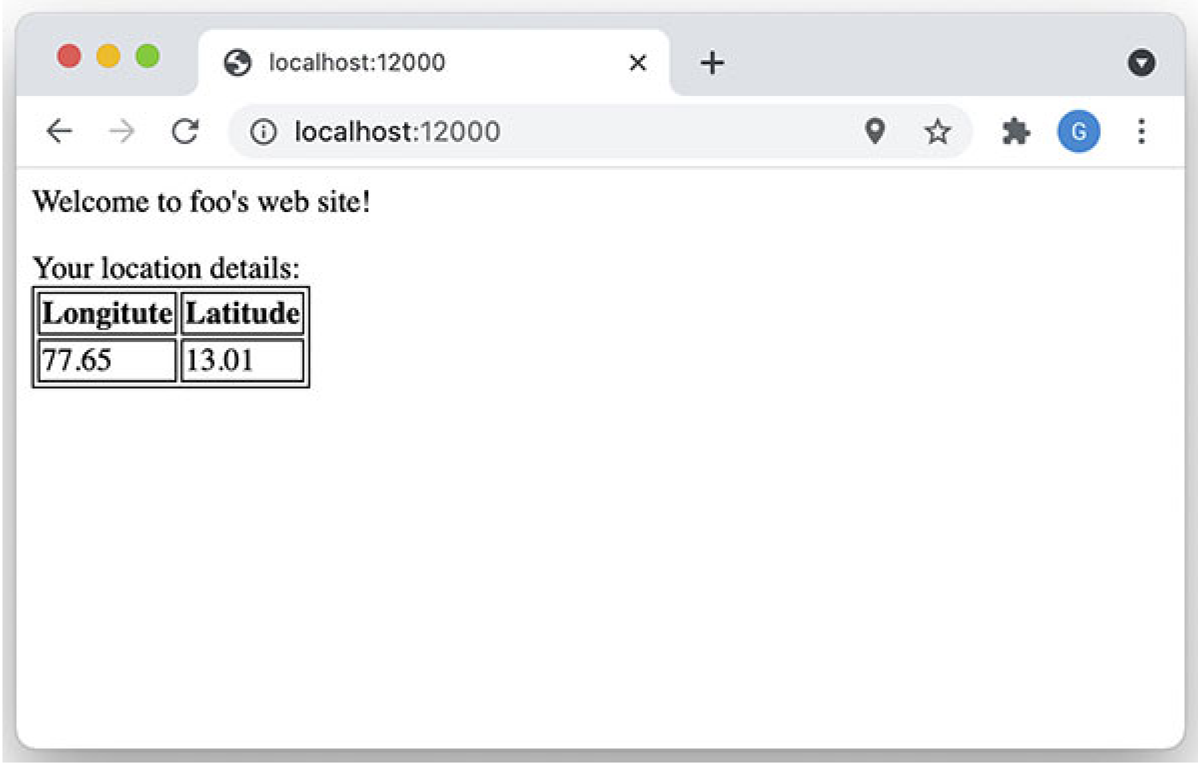

Figure 8.44: *A web page with geolocation*

Authentication

User authentication is a complex process. On one side, a website has to deal with the algorithms that define passwords, while on the other hand, it needs to take care of managing its safe creation, transport, validation, and processing.

The complexity increases with federated accounts, wherein websites reuse another account that is already created or one that is already logged into.

The complexity increases further with two-factor authentication for tightening the validation methods, as there are workflows that include multiple validation methods.

Non-standard credential management APIs address most of these complexities. Web Authentication APIs, which is an extension of Credential management APIs, expose interfaces for creating and obtaining new or existing credentials associated with a user account. The APIs take care of preventing common security threats like phishing and password attacks.

While it is possible to authenticate using plain Node.js APIs, it is neither straightforward nor practical, so a recommended approach is to use third-party libraries that specialize in authentication feature.

Admin dashboard

Websites that deal with commercial transactions can have an administrative dashboard that shows multiple views around the usage of the website, through the mainstream path. Here are some examples of what these dashboards can typically include:

- **Tickets**: open, closed, new
- **Users**: total users, daily, weekly, monthly
- **Resource usage**: CPU, memory, network
- **Geography**: NA, EU, EMEA, AP

The following screenshot shows a typical website admin dashboard:

Figure 8.45: *A web page with sample admin dashboard*

User profile

A user profile page is a form that collects users' information beyond just the username. This information can be useful for the site owner for associating the site activities and customer's attributes to glean valuable insights. On the other hand, the user benefits through the profile in the form of customized and personalized experience in the website, which has taken into account the extended personal information while processing the user request and composing the web page from the server response.

The following code shows a typical user profile page:

Update your profile

First name

Last name

Email ID

Twitter handle

Profile picture upload

Figure 8.46: *A web page with user profile form*

Conclusion

In this chapter, we bridged the knowledge of our backend with that of the frontend and filled the missing piece in the bigger picture of web architecture—frontend design considerations and web page elements. We looked at the common requirements of a website (frontend rendering) and how pages and forms can implement some of those common features. First, we looked at the user experience of website and how that drives the considerations of web page constitution. We also looked at the various

constituent elements that contribute to a website based on different use cases and how special components meet those requirements in the page design.

This completes our end-to-end study of developing a web server and website using just Node.js and command line interface.

In the next chapter, we will examine non-functional factors that are important in terms of hosting an industrial-strength and production-grade website. These factors include deployment topologies; scaling considerations; **Reliability, Availability, and Serviceability** (**RAS**); and monitoring and tracing. These elements help execute the life cycle operations of our software (such as development, testing, deployment, upgrade, and support) with relative ease and maintain our website with enhanced confidence.

CHAPTER 9

Making Our Website Production Grade

With the previous chapters, we completed our end-to-end study of developing a web server and website using Node.js and command line interface as the sole tool from the perspective of functional capabilities. In this chapter, we will examine the non-functional factors that are important in terms of hosting an industrial-strength and production-grade website application. These factors include deployment topologies; scaling considerations; **Reliability, Availability, and Serviceability** (**RAS**) features; security; observability; and documentation. These ingredients help execute the life cycle operations of our software (such as development, testing, deployment, upgrade, and support) with relative ease and maintain our website with enhanced confidence on its usability. While this knowledge is not an absolute must for building a sample website, we would consider it critical for a commercially successful product.

Structure

In this chapter, we will cover the following topics:

- Reliability
- Availability
- Scalability
- Observability
- Security
- Documentation

Objective

After studying this chapter, you will be able to understand the enterprise enablement of a software in general and a web application in particular. You will understand how our website can contain the complexities when the consumption rate increases to enterprise grade and consumption nature become versatile and heterogeneous. You will also learn how the application incorporates numerous best practices to make it more robust, secure, serviceable, and observable.

Enterprise enabling components

The following diagram represents how the above-mentioned attributes help strengthen a software application for attaining enterprise grade:

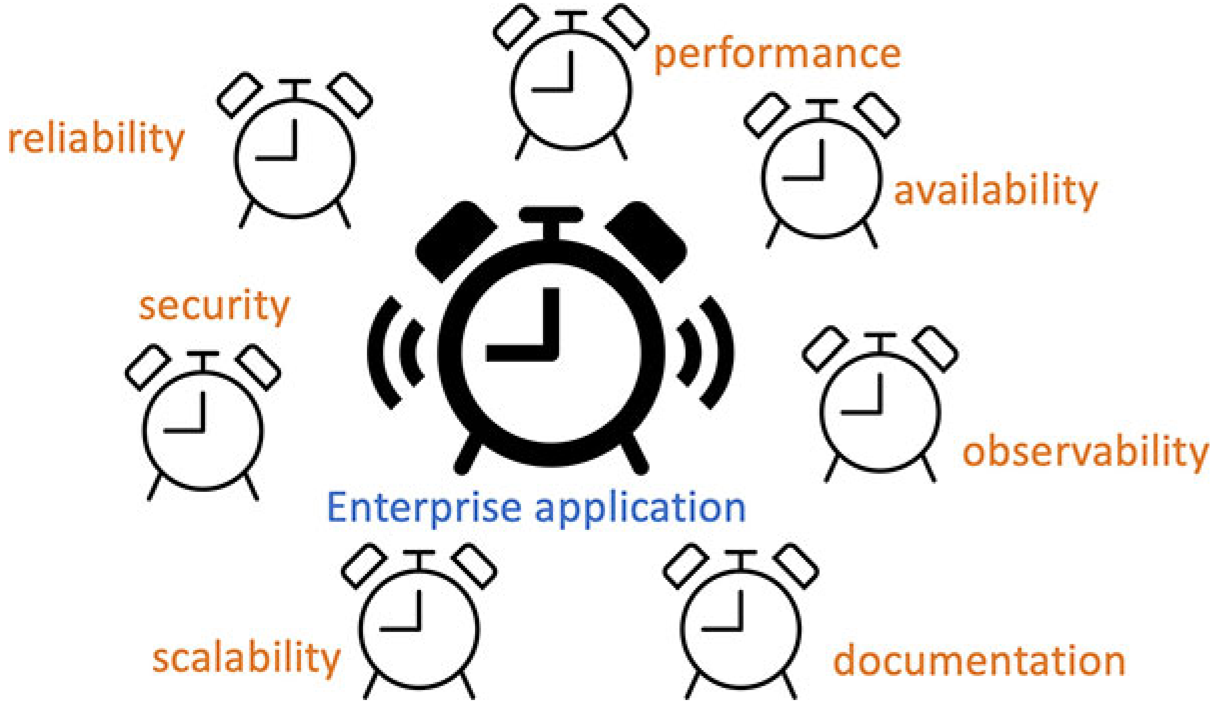

Figure 9.1: *Constituent elements of a high workload application*

Performance

A web application's performance (responsiveness) is important for its success, so we want to discuss it in depth, covering the application and the rest of the execution environment. For that, we focus on the performance aspects in the next chapter, covering both the web application and the Node.js runtime. So, we will pass on performance here.

Reliability

Reliability is a software's ability to perform its stated capabilities as per the specification under all reasonably possible execution environments.

The following equation can be used to mathematically represent the degree of reliability:

$$R = e\text{-}\lambda t$$

Here, R is the reliability, t is the time period in which the reliability is measured, and λ is the failure rate within the specified interval. From the relationship, it is evident that the reliability increases for increased time periods of constant failure rates. Also, the reliability decreases when the failure rate increases for a constant period.

Reliability is largely a measure of the said software's quality. This is because a highly reliable software will cover many possible scenarios in which the software will be used. So, any action performed to improve reliability is, in a way, improving the software's overall quality.

The reliability of a web application is the sum total effect of the reliability of its constituent components: the frontend and backend, and its internal and external services. In addition to these, reliability depends on the way these components interact and the transportation of the input and output of the business data. So, care needs to be taken to ensure that the component interaction scenarios are covered when testing for reliability, not just unit tests or component-based testing.

The following table shows the reliability matrix for a web application—a minimum checklist that, when met, assures a reasonable level of reliability:

Reliability	Fault anticipation	Fault interception	Fault toleration
Frontend	✓	✓	✓
Backend	✓	✓	✓
Services	✓	✓	✓
Network	✓	✓	✓

File system	✓	✓	✓

Table 9.1: *Web application reliability matrix*

In other words, these components should ensure an acceptable degree of fault anticipation, interception, and toleration to make the overall web application reliable.

In this chapter, we will look at the most common reliability issues that a web application is subject to and illustrate the remediation or best practices to be followed in such scenarios. We will start from the frontend, go backward in the application topology, and cover all the components and their interactions in a specific order, roughly following the request-response trajectory.

Issue

The client view of port's size and shape appear different when the application is accessed through different devices. Also, the resize behavior is not consistent.

Remediation

The view port meta tag in HTML can be used to define the overall characteristics of the view port. And more specifically, the width is defined as a function of the device's width and a scaling factor to set the default as the initial window width. The scaling factor is applied onto the width of the viewport thus obtained.

The following code is an example of how to set the scaling factor correctly:

```
1. <meta name="viewport" content="width=device-width, initial-scale=0.5">
```

In this example, half of the device's width is used to define the width of the viewport. So, the actual width becomes a function of the actual device independent of any hard-coded or development-time heuristics.

Issue

Form fields and controls are not scaling relative to the window size, and the alignments of form fields and controls are haphazard.

Remediation

Cascading Style Sheets (CSS) present very powerful semantics for positioning and aligning form controls within a form and defining the resize strategy when the form resizes.

Issue

Inconsistent, irrelevant, and insecure error messages appear on the page when the backend crashes. The following figure shows an example of a bad error response:

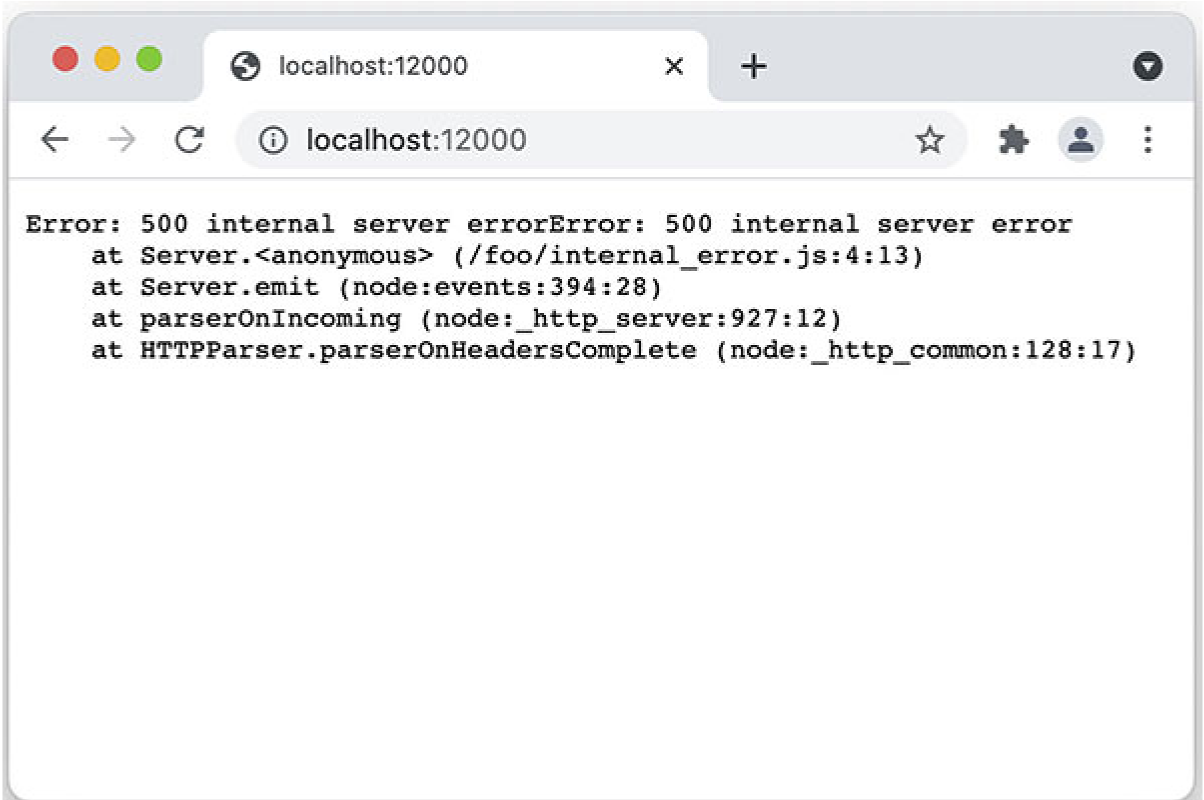

Figure 9.2: *Example of a bad error message*

Remediation

Ensure that all responses that reach the client are hand-crafted as opposed to funneled directly from the error stack.

Ensure that all responses that reach the client are meaningful and complimented with the standard HTTP status code that most reasonably reflects the error scenario.

Sometimes it is too tempting to intercept an error and terminate the request by providing the response at the error site itself. The site may be too deep into the server's code (for example, an internal service invocation), and throwing that error message to the user as is will be less useful. In such cases, the response should be abstracted to a certain level for the user to relate to it and potentially take remedial action.

On the other hand, generic error messages (such as "*some error occurred, sorry for the inconvenience*") are not useful either. So, the bottom line is that the responses should be well abstracted to ensure that they retain the meaning without revealing the internal specifics.

Ensure that all responses that reach the client are audited well from the security point of view, that is, they don't reveal the internal details of how the server functions or what data elements it processes. As a best practice, always return error codes and follow the standard.

Tip: Error messages in a web application should be carefully crafted. Client errors should be well described, directed, and helpful for the user to fix by themselves intuitively. On the other hand, server errors should abstract the actual error with a high-level error message, hiding the internal functioning of the server while providing a meaningful message to the user, more specifically the message: should the user retry again immediately, retry after some time, or wait for a notification, and so on.

Issue

Form error management is poor. It reports one error at a time and goes round in cycles.

Remediation

Handle form validations with extra care, especially with large forms. Clearly define what is mandatory and what is optional. Define the data type of form fields properly, more specifically the date fields, mobile number fields, and such where multiple formats are possible. Clearly specify what

format is accepted, with example, if possible. Accept the most common formats without being strict, if possible.

Show all errors upfront instead of hiding them from the user until they click the form submit button.

Don't clean up the properly filled fields upon validation. Depending on how large the form is, these can cause severe usability issues as well.

A bad example of an error message: "*you might have pressed the back button in the browser or might have refreshed the page, or some other unknown error occurred!*". What is the message to the end user here? It's that the software is unable to understand what is happening, which leads to poor end-user experience.

Issue

Captcha is useless—neither readable nor consistent.

The following figure shows an example of a bad captcha that is neither readable nor meaningful:

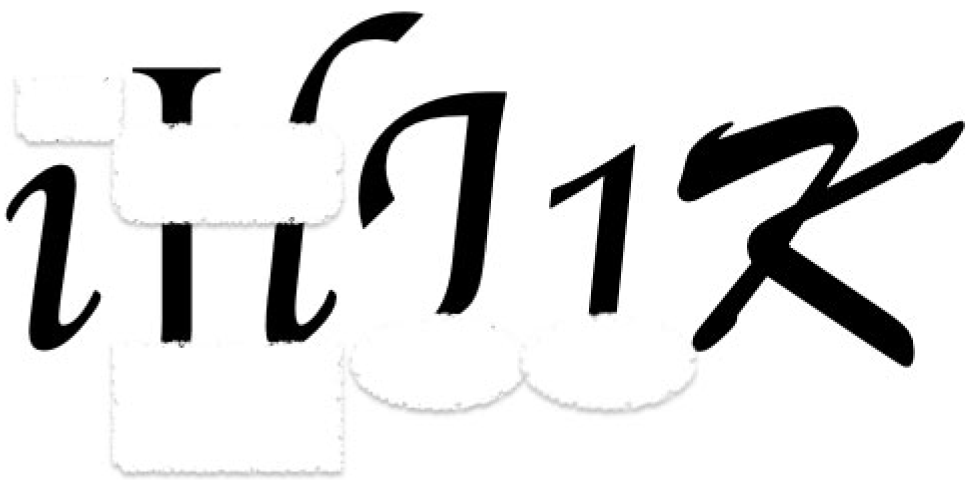

Figure 9.3: *Example of a bad captcha*

Remediation

If captcha is used, be consistent with its font and range. Some captchas are confusing. For example, an 'F' and a 'T' will look the same in some bad captchas, which can add to user frustration.

Issue

Inconsistent behavior with sessions—arbitrary persistence across sessions. Additionally, the session ID hints at the business logic of the application.

The following figure shows an example of a poorly crafted session record, as the key-value pair is revealing the business logic of the web application:

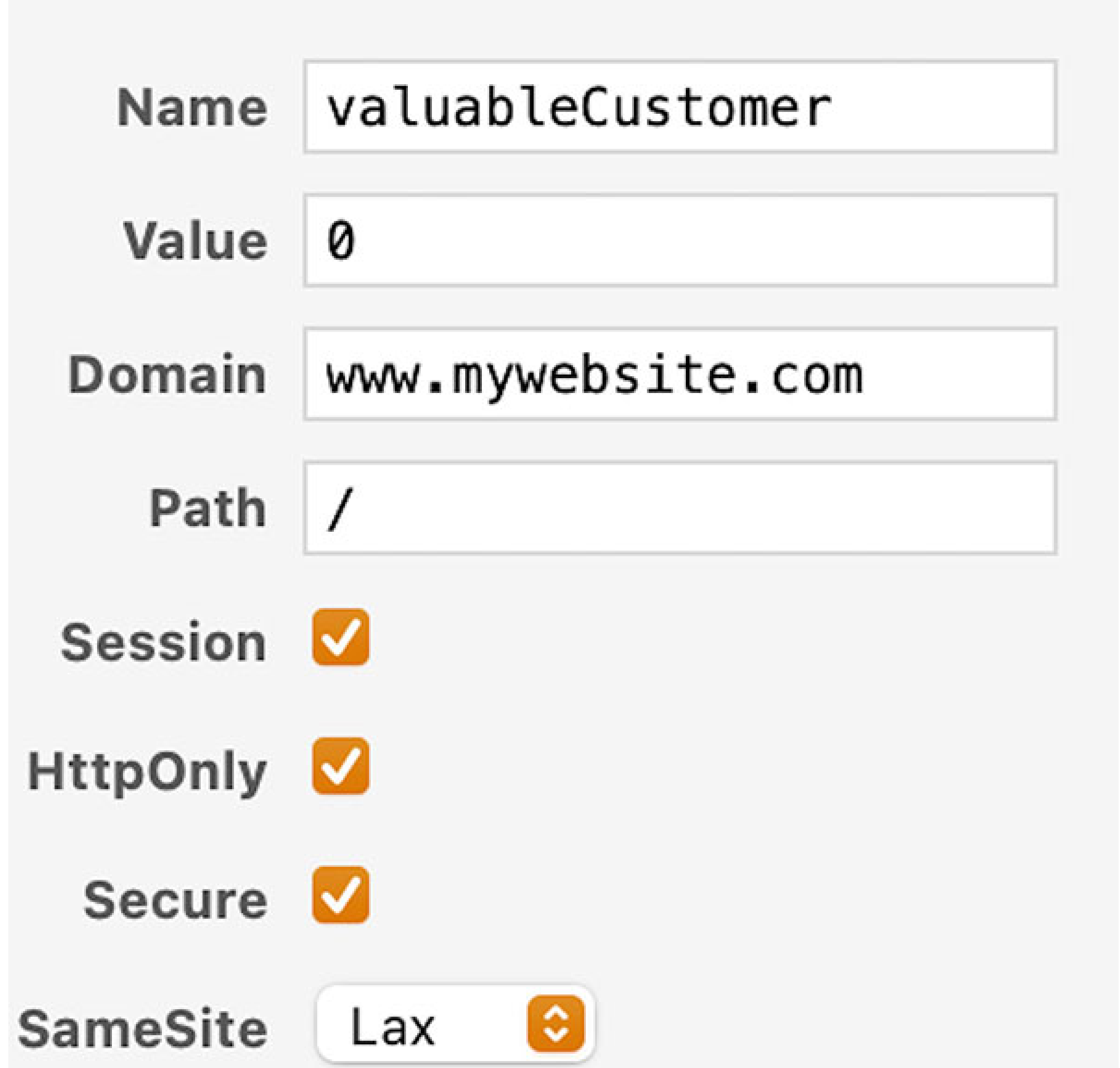

Figure 9.4: *Example of a poor session cookie object*

Remediation

Implement session persistence separately with the main backend process; a database implementation that adheres to the data integrity constraints is

preferred. That way, sessions become immune to server crashes and can be used consistently.

Also, ensure that the session IDs (that are stored on the client side) are only symbolic names and don't reveal the real meaning of the data that they back up. Additionally, make the session ID long enough to be immune to brute force guessing by attackers.

Ensure that all the state transitions in the application are properly covered under the session's life cycle and there are no loose ends. An example is a specific code flow that terminates the current request and completes the response with the user logging off but the session is still valid due to an unforeseen flow path in code that the session did not cover.

Issue

File upload function is flaky. Huge files cause the server to hang, and some binary files can crash the server.

Remediation

Ensure that the file upload feature is properly sanitized in terms of valid file types and length. Use the file content filter as well as size filter to abort outlier cases. Throw proper error messages for such cases.

Issue

Based on the location and context of the server failure, the messages appear differently and in a non-deterministic manner. This inconsistency affects the interaction with the server.

Remediation

This happens when broken data access leaves the service in an inconsistent state. Ensure that your application services are not managing the data directly and are delegated to database management systems instead.

Ensure that the database offers transactional integrity to the data it hosts.

Ensure that the data caching decisions are consistent across different services.

Issue

Page loads are inconsistent. Some take less and some take more, based on the request. Some interactions result in timeout or a 404 error.

Remediation

Avoid tight coupling between your application's internal and external services, that is, establish inter-service communication through well-defined API calls, make those calls asynchronous, and define fault tolerating mechanisms (error handling sequences in simple words) on the service invocation sites to handle service invocation failures.

Issue

The process is seen to be slowing down over time. When inspected with tools, it is seen as holding a large amount of native memory.

Remediation

Native add-ons are the first suspects that use native memory outside of the JavaScript heap. Avoid allocating a large amount of native memory that is unmanaged by the virtual machine, or it can potentially affect the reliability of the process in general.

Issue

Inconsistent payment behavior—gateway failures cause lack of determinism.

Remediation

Ensure that the gateway function is sufficiently robust, atomic, and flexible, and specifically manage all the failure cases appropriately. Payment function is very sensitive to the user, so implement it after sufficient and surplus testing.

Availability

Availability is the ability of a software to be available to perform its stated capabilities in the eventuality of faults or erroneous situations.

It is defined as the duration for which an application's service is available, in relation to the total duration in which the measurement was taken. Availability is often considered complimentary to reliability because a software's availability coupled with its reliability leads to its robustness, which is a critical quality of production systems.

The following equation can be used to mathematically represent the degree of availability:

$$A = \frac{tf}{tf + tr}$$

Here, *tf* is the time to failure and *tr* is the time to recover, on an average, and *A* is the average degree of availability, measured in the time span within which *tf* and *tr* are computed.

Listed here are the most common availability issues and their known solutions and best practices:

Issue

The web application crashes in certain use cases due to bad or malicious input.

The following figure provides a typical example of a server crash due to a simple bug in the program in one code flow path:

```
01. Error: write EPIPE
02.     at WriteWrap.onWriteComplete (internal/stream_base_commons.js:92:16)
03. Emitted 'error' event on ClientRequest instance at:
04.     at Socket.socketErrorListener (_http_client.js:461:9)
05.     at Socket.emit (events.js:315:20)
06.     at emitErrorNT (internal/streams/destroy.js:96:8)
07.     at emitErrorCloseNT (internal/streams/destroy.js:68:3)
08.     at processTicksAndRejections (internal/process/task_queues.js:84:21) {
09.   errno: -32,
10.   code: 'EPIPE',
11.   syscall: 'write'
12. }
```

Figure 9.5: Example of a server crash

Remediation

Ensure that the application in general and the most common request handler modules in particular have exception handlers installed, apart from the error handlers on the participating objects. The most common programming model of Node.js is to have an '*error*' event defined on the I/O routines (such as request and response).

The following code illustrates how to install an error handler on the participating object:

```
1. req.on('error', (err) => {
2.   log(`request encountered an error: ${err}`)
3. })
```

However, the implication of not implementing an error handler is that the error gets ignored, which may cause the final outcome of the request to become obscure. On the other hand, exception handlers (such as try-catch blocks) are part of general programming model and help catch unexpected programming scenarios.

The following code snippet illustrates how to install an exception handler:

```
1. try {
2.   const ret = await send(data)
3. } catch (err) {
4.   console.log(err)
5. }
```

The implication of not implementing an exception handler is that the process terminates if an exception occurs, and there is no handler to absorb it.

So, while not having an error handler only affects the serviceability of the application functions, not having an exception handler affects the availability of the application itself. So, a best practice is to cover vital parts of the application with an exception handler. This ensures that faulty transactions are aborted with proper exception messages, while good

transactions continue to be processed, leading to improved availability of the application.

When dealing with exceptions, ensure that there is a designated catch block to handle every explicit exception thrown from the application. There are exceptions that can leak through it (for example the exceptions that the runtime API throws), but we should handle as many exceptions as possible.

A Node.js-specific best practice is to emit errors as opposed to throwing exceptions wherever possible. When the erroneous context has an event handler object as a subject of the error or the consumer of the error, it is reasonable to emit the error on that event handling object.

The following code snippet illustrates how to effectively use event mechanism to intercept and propagate erroneous situations in your program:

```
1. setTimeout(() => {
2.   if (source.hasData)
3.     source.emit('error', 'ETIMEOUT')
4. },10000)
```

Issue

The web application crashes/terminates under certain unavoidable and unanticipated scenarios.

Remediation

Despite the earlier precaution being accommodated, the application can abruptly terminate at times. An example is the lack of resources to process the current load (such as memory, thread, and file descriptor).

On the eventuality of such a crash, the simplest strategy in a production box is to collect the crash artifacts (such as dumps, traces, and logs) and then immediately restart the application. But there is a definite delay, which can range from minutes to hours (depending on how fast the crash is detected) if this is a manual process.

So, a recommendation is to run the application through a launcher script that can run a loop, whose sole function is to:

- Run the application

- Collect the must-gather artifacts
- Repeat this in a loop

A simple pseudo UNIX shell script is shown here to illustrate how to carry this out:

```
1. while [1]
2.   do
3.   node app.js
4.   if test core
5.     then
6.     mkdir /tmp/`date +"%m%d%y%H%M%S"`
7.     mv core /tmp/`date +"%m%d%y%H%M%S"`/core
8.   fi
9. done
```

The preceding code runs the Node.js application in an infinite loop. In each loop iteration, we test for the presence of a core file by name after the application is terminated (potentially due to a crash), and if yes, we create a temporary folder with the current date and time and move the core file to that location. Then, we rerun the application. This ensures that the core files are not lost in the eventuality of server crashes.

This will also ensure that the application is always available at a granularity equal to the startup time of the application. But it also has a side effect—the request that actually caused the crash, and all other requests that are running concurrently in the application, will be aborted together and need to be re-issued from the client side.

> **Note:** Do not compromise reliability and functional correctness for availability, that is, do not try to cover up or circumvent an abnormal program scenario that is completely unanticipated by the application and/or the runtime. Doing so by ignoring the error and trying to continue can lead to more severe side effects for the application.

Issue

We cannot afford the aforementioned scenario of call-drops and low availability at the time of server crashes.

Remediation

If the above setup does not meet your application's **Service Level Agreement** (**SLA**), the next action is to make redundant copies of your application in multiple systems. In simple words, this will make multiple, identical copies of your application running in different systems, all ready to serve the client. Redundancy is a production best practice process that recommends creating systems with improved degrees of availability.

There are several approaches to achieve high availability through redundancy, with varying costs to maintain availability tradeoff.

Master-slave replication

In the master-slave model, we run the main server in primary mode, while the slave system will be on standby. When the master crashes, the slave rises up to the occasion and manages the load, designating itself as the master. Behind the scenes, the master will bootstrap from crash and become the new slave. When the new master crashes, the new slave becomes the master, and the cycle goes on.

The following diagram illustrates the master-slave model in availability architecture:

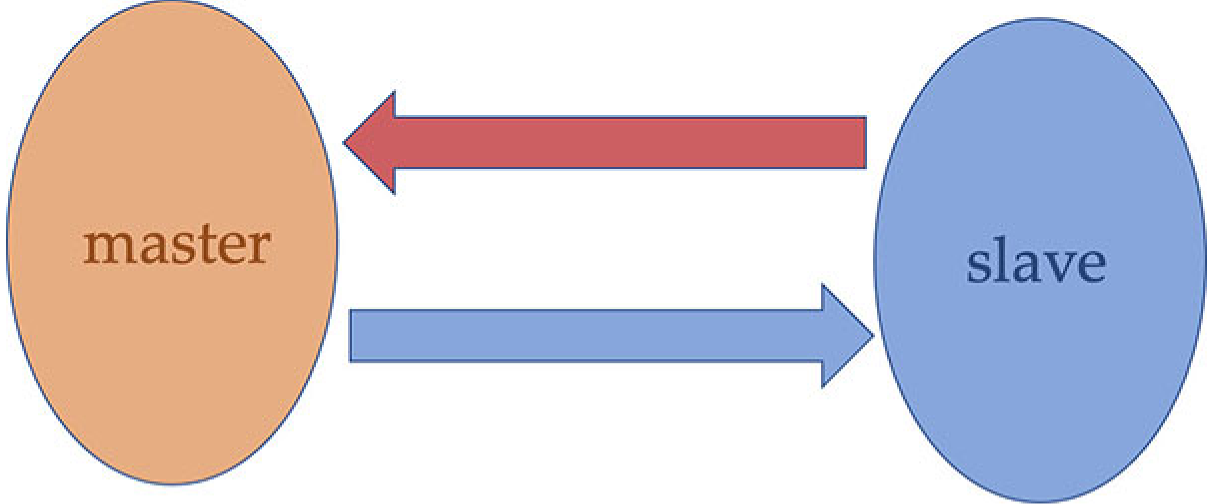

Figure 9.6: *Application replicated in a master and slave system*

The tradeoff here is the low maintenance cost for a little fallback on the availability. There is still a time gap between the master becoming unavailable and the slave becoming available.

Peer-peer replication

In the master-slave model, the key drawback was the downtime between master crash and slave boot-up. In the peer-to-peer model, both or all the systems are ready to serve clients at any point. This model banks on the statistical theory that the probability of all the systems to be down simultaneously is almost zero.

The following diagram demonstrates the peer-peer model in availability architecture:

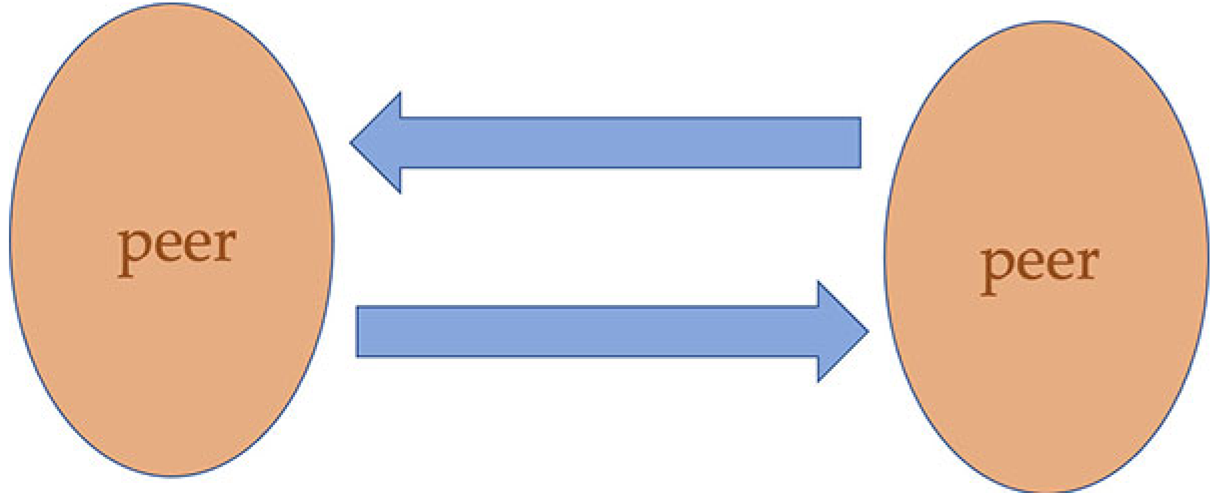

Figure 9.7: *Application replicated in two identical peers*

The tradeoff with peer-peer system is the high cost to maintain all the systems, but it ensures a high degree of availability. Also, there is synchronization between the peers: how to determine which request goes to which peer?

Cluster and load balancer

The concern of peer-peer is addressed in the cluster with load balancer topology. In this, several identical servers (called replicas) run parallelly while the request routing is carried out by a load balancer (also called reverse proxy) placed between the client and the peers (also called replicas).

This solves the issue of synchronization. The load balancer performs a number of other actions, such as page caching and request validations.

The cluster-load balancer model in availability architecture is illustrated as follows:

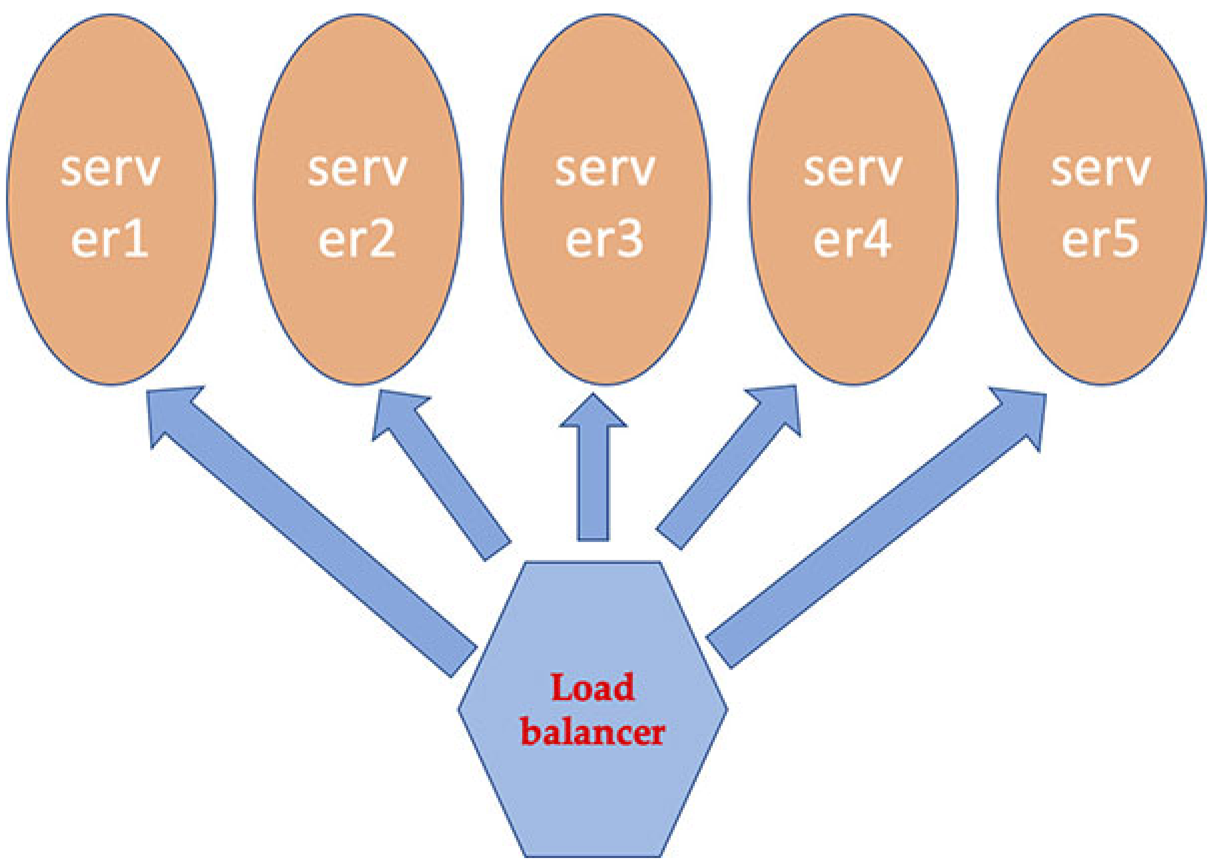

Figure 9.8: *Application replicated in multiple systems*

Issue

There is still a concern on the availability, as the systems (nodes) in the cluster are physically co-located and incidents like fire can cause the entire system to be damaged and thereby, unavailable.

Remediation

The recommended solution for this is to diversify the application with respect to its physical presence, that is, have data centers and clusters at different geographical locations.

The following diagram illustrates the cluster-load balancer with a geographically distributed model in availability architecture:

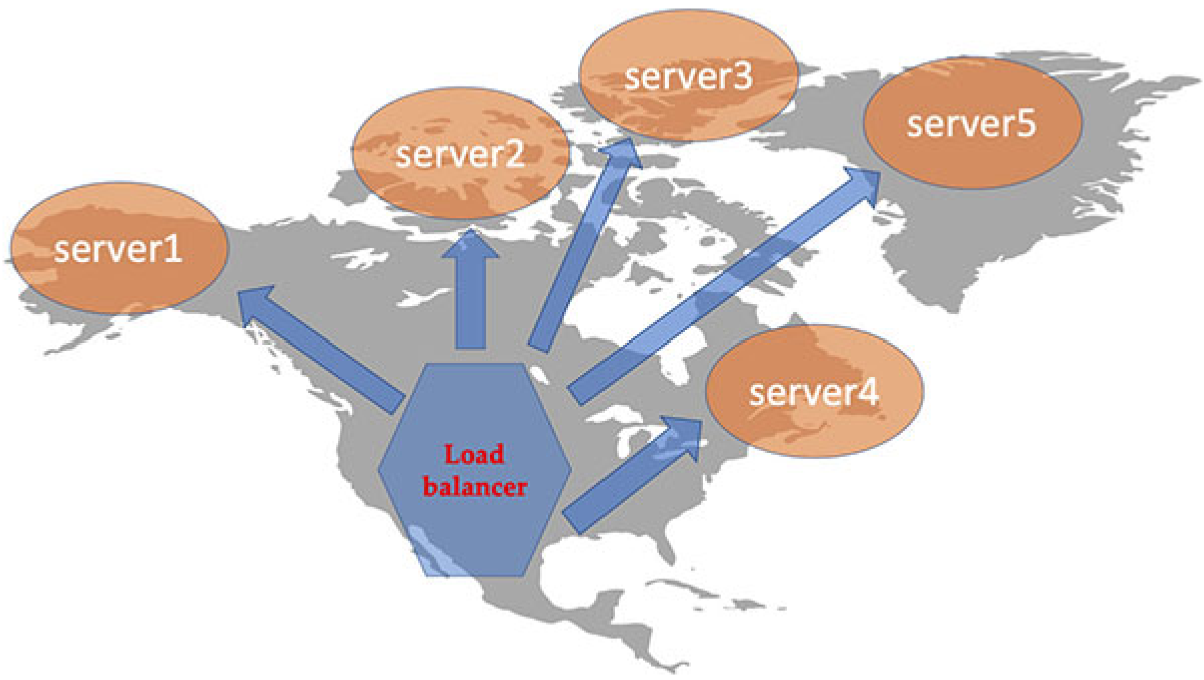

Figure 9.9: *Application replicated in multiple geographical locations*

Note: Application availability with multiple servers installed across multiple geographical locations and synchronized by a central router to balance the load is the most modern format for achieving high availability at the time of writing this book.

Issue

The above-mentioned methods work well for the application and its dependent services. However, the database that the services connect is still a single instance and cannot be replicated the way the application is. So, all the mentioned issues affect the database too. This leads to the reduction of overall application's availability, as there is at least one component still vulnerable from the high availability standpoint.

Remediation

Databases can be scaled up as well, through a technique called **sharding**. A shard is a subsidiary database of the main database that is partitioned in terms of rows. With sharding in place, several shards will constitute the

logical aggregation of a database. Database operations scoped with rows (deletion, update, insertion, and so on) can be performed on any shard, while operations scoped at the table level need aggregated views of all the shards. These internal complexities are managed by the database server software and are fully concealed from the consuming application.

The following is an illustration of the database replication model in availability architecture:

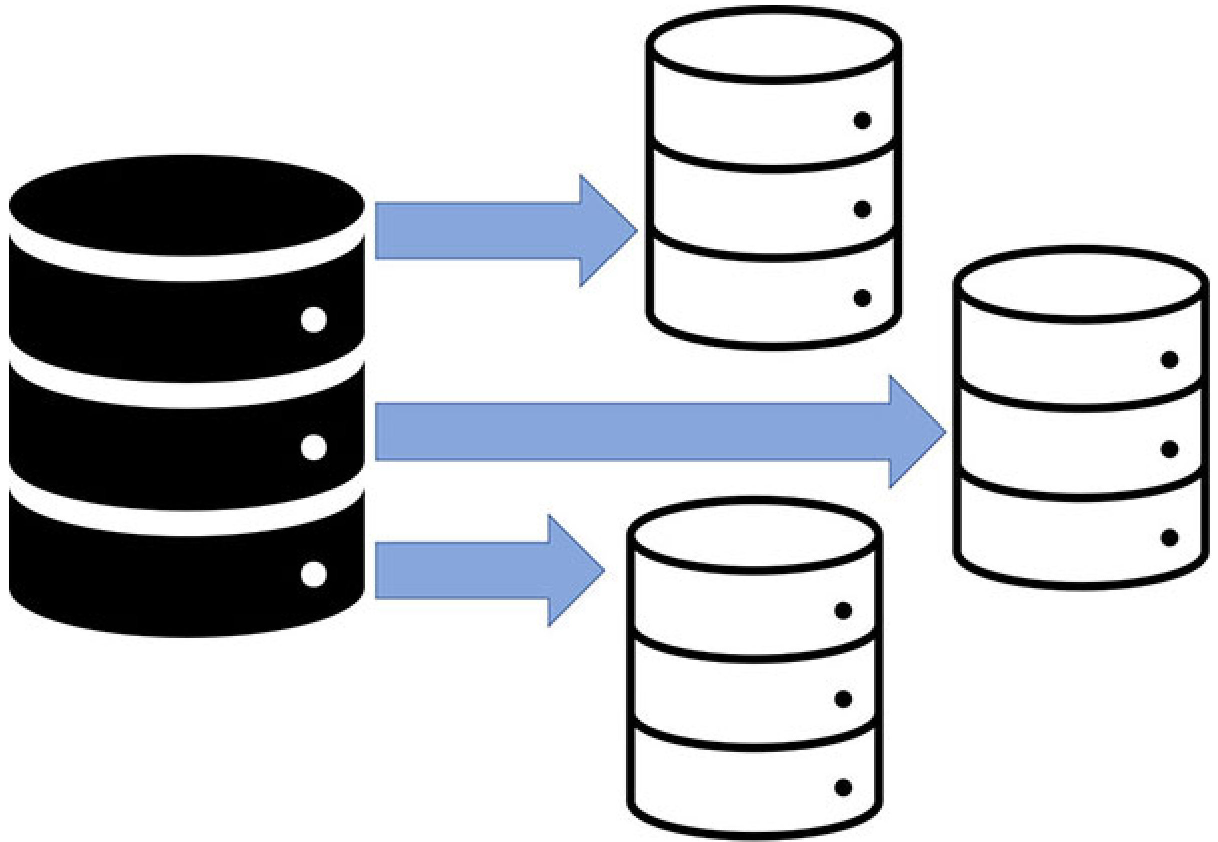

Figure 9.10: *Database sharding*

Additional database-related best practices include performing regular data backups and regular data replication.

Issue

Despite all these features, there are times when the application needs to be brought down for maintenance, for example, to upgrade the server application. This is an unavoidable downtime and affects the application's availability.

Remediation

There are techniques to incrementally upgrade the application in a controlled manner. One of these techniques is called blue-green deployment. Here's how the blue-green deployment works:

1. Part of the instances is brought down
2. The brought down instances are upgraded to a new version
3. The traffic is diverted to old instances in the meanwhile
4. The brought down instances are made up and running
5. Some traffic is diverted to the new instances
6. Once the new version stabilizes, more traffic is diverted
7. This is repeated until all the instances are upgraded

Another approach that works for certain runtimes and languages is hot code replacement. In hot code replacement, the running application is amended even without it being brought down. Instead, special instrumentation APIs are used to modify the code and data—classes, objects, methods, fields, configurations, strings, and so on. This way, the application is upgraded in process.

Scalability

Scalability is the ability of a software to perform its stated capabilities under growing load, without compromising on reliability, availability, and performance.

Bringing the definition to the context of a web application, scalability is the ability of a web application to adjust with the change in traffic and provide the same response time for all clients.

Scalability can be measured as a ratio of the application's performance (average response time) to the increase in the client request count.

The following equation can be used to mathematically represent the degree of scalability:

$$S = \frac{Req(t)}{Res(t)}$$

Here,/ *Req(t)* represents the request density (the number of incoming requests within a specific time span) and *Res(t)* represents the average responses sent in the same time span.

At high level, scalability has two governing principles:

- Add more resources (CPU, memory, I/O) to handle the increased load
- Reduce bottlenecks (single point critical sections) as much as possible

The key here is to be able to supply server resources in proportion to the increase in client demand and sustain the ability of these resources to function independently. A static configuration of resources that is pre-defined based on the perceived amount of load is a good baseline for a healthy application. An application capable of continuously adjusting these resources as the workload characteristics change qualifies as a healthy and scalable application.

All the scalability best practices specialize on the two above-mentioned elements.

Node.js as a language runtime is designed to be scalable, which means, there is a high level of resource autonomy—the ability to create and use resources without any side effects to the environment— at the platform and the API level. The following are the main reasons for this are:

- Single-threaded virtual machine
- Event-driven architecture with asynchronous programming

The single–threaded programming model ensures that there is no shared data and locks in the program. At the application level, this would mean that the increasing number of concurrent client requests does not impose any bottleneck to the application or process. The opposite of single–threaded programming model is multi-threaded or a thread–pool model, wherein the concurrency is managed by multiple threads, with associated resources and shared data that all the threads synchronize upon.

Event–driven architecture ensures that the execution density is maximum at any given point. This means the single thread is free to perform queued operations while the events (mostly I/O events) are processed behind the scenes.

Despite these benefits of inherent scalability, we must be aware of several considerations for a web application and address them through best practices.

Note: Execution density is defined as the number of operations performed per second at the lowest level of measurement possible. This mostly boils down to the average CPU usage, as active usage of the CPU is a true measure of how much of the allocated time of a specific CPU to a specific thread was leveraged by the thread.

Vertical scalability

Vertical scalability is the ability to effectively increase the resources in response to an increase in the load from within the server's system. In other words, it is how the application can withstand the increased load, by using more CPU, memory, network resource, and so on.

The following diagram shows stacked up CPUs in a single system to represent the approach of vertical scalability:

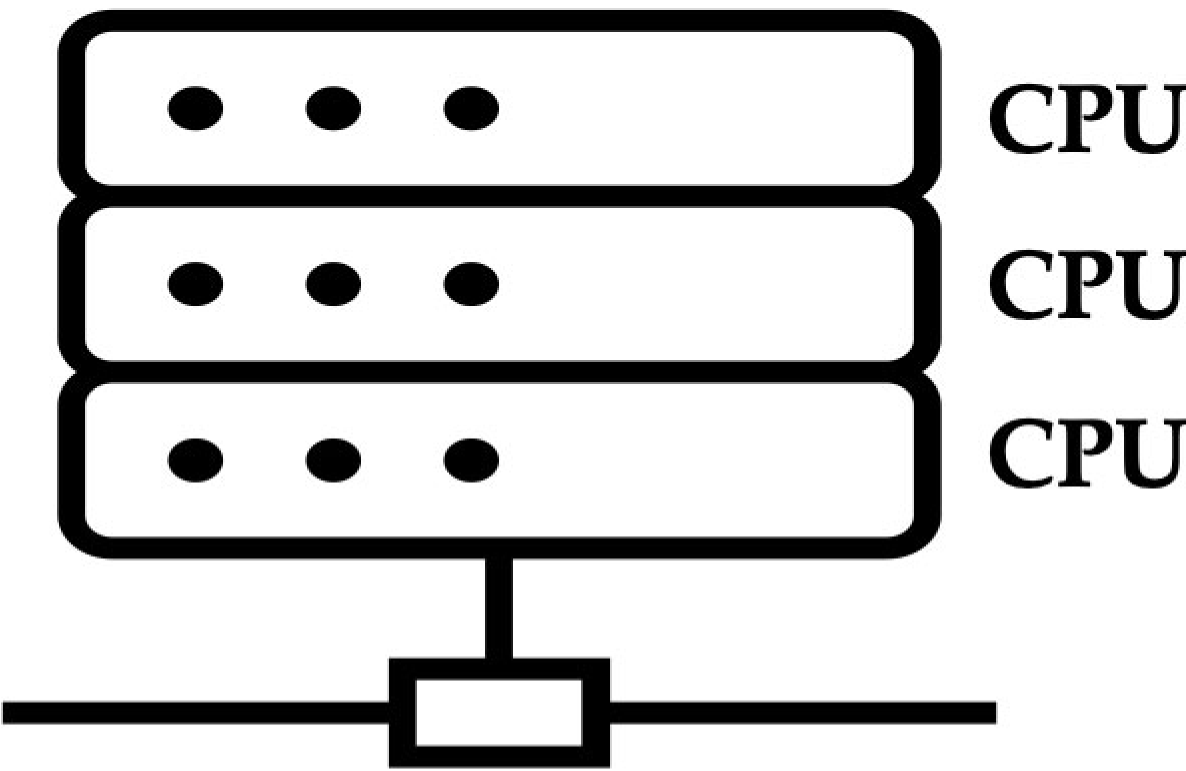

Figure 9.11: *Representation of vertical scaling*

Issue

My system has several CPUs, and yet the Node.js application is not exercising all of them. On the other hand, a performance saturation can be seen when the concurrent client count increases beyond certain limits. As a result, the system is incapable of handling the load, while the available resources are underutilized.

The following graph illustrates a poorly scalable system, wherein the throughput drops after a certain number of (1500) concurrent users connecting to the application instance:

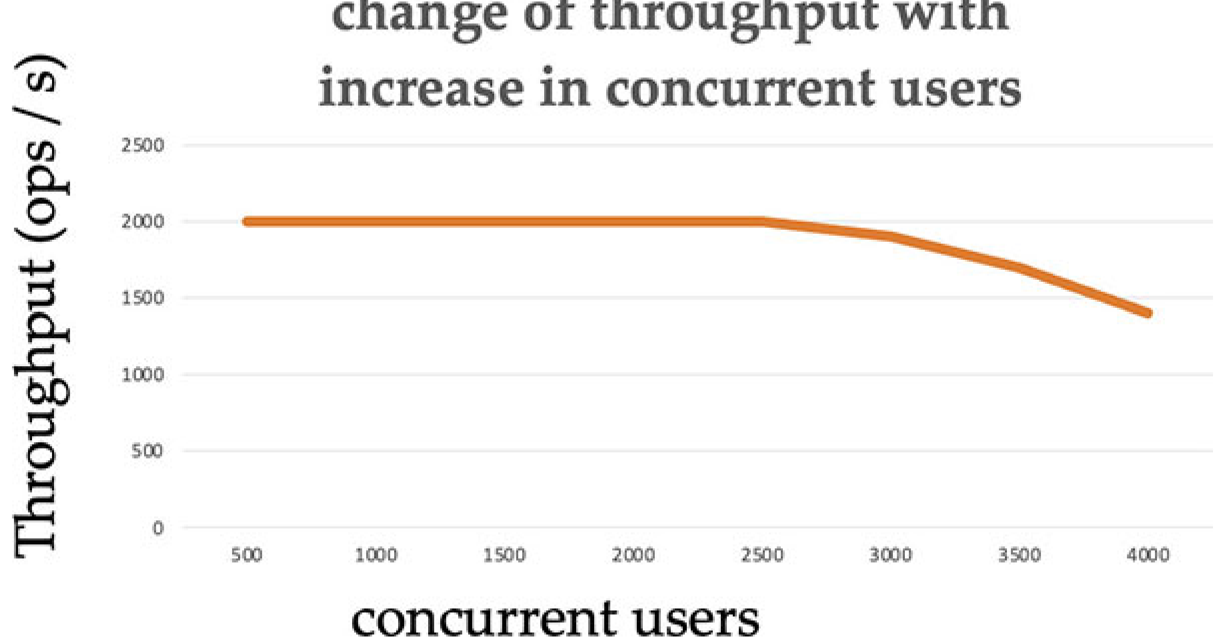

Figure 9.12: *An example graph showing performance saturation*

Note: Performance saturation is a phenomenon in which the throughput reduces slowly in response to an increase in concurrent workload.

Remediation

This is a known side-effect of the single-threaded programming model that Node.js presents. As we are increasing the concurrency level to extreme scales in one thread itself, the model does not go hand-in-hand with multi-processor systems (systems that are largely developed to support multi-threaded software).

We need to realize a few things before even recognizing this as a problem/tradeoff. Some key aspects around Node.js concurrency are explained ahead. Evaluate your workload characteristics and use case carefully in the context of Node.js's concurrency before applying scalability best practices.

Node.js is NOT single-threaded!

"*Is Node.js single-threaded?*" This is a recurring question, like the chicken-egg puzzle. In many places (even in this book), we read that Node.js is single threaded, but many other places talk about threads and (even)

threading in Node.js. What is the truth? Let's examine a representative Node.js process from the operating system's perspective. For this, let's perform this simple activity:

1. List all the processes running in the shell, expanding on the process threads
2. Start a trivial Node.js process
3. Make it run in the background
4. Repeat the first step—list the processes and thread in the shell
5. Count the number of new threads that have come up

The following screenshot shows that Node.js is not a single-threaded process by first printing the number of threads in an empty shell, and then counting the number of new threads seen through the shell with a Node.js process started:

```
$ps -eL
  PID   LWP TTY          TIME CMD
    1     1 pts/0    00:00:00 bash
   40    40 pts/0    00:00:00 ps
$node
Welcome to Node.js v16.5.0.
Type ".help" for more information.
>
[1]+  Stopped                 node
$ps -eL
  PID   LWP TTY          TIME CMD
    1     1 pts/0    00:00:00 bash
   41    41 pts/0    00:00:00 node
   41    42 pts/0    00:00:00 node
   41    43 pts/0    00:00:00 node
   41    44 pts/0    00:00:00 node
   41    45 pts/0    00:00:00 node
   41    46 pts/0    00:00:00 node
   41    47 pts/0    00:00:00 node
   41    48 pts/0    00:00:00 node
   41    49 pts/0    00:00:00 node
   41    50 pts/0    00:00:00 node
   41    51 pts/0    00:00:00 node
   52    52 pts/0    00:00:00 ps
$
```

Figure 9.13: *Multi-thread view of a Node.js process*

So, it is not single-threaded and has more than 10 threads inside! What does this mean?

Here are the facts about threading in Node.js. Node.js is technically multi-threaded, which means the virtual machine or the language runtime has multiple threads in it. However, the application runs on a single thread. The other threads support tasks that are essential for the non-blocking, asynchronous, event-driven programming model. Here are some examples:

- Perform a disk I/O (that was requested by the application) in a separate thread rather than in the main thread (also known as the application thread).
- Run garbage collection concurrent to the application. Evidently, these operations take a lot of CPUs and can potentially delay activities in the main thread if carried out there.

In summary, Node.js runs its application on a single thread, but there are several supporting threads to implement the high level of concurrency.

So, if you have machines with 2, 4, 8, or even 16 logical CPUs, you can host a single Node.js process without worrying about resource underutilization. A concurrent workload will ensure that most CPUs are engaged in a reasonable manner.

Efficiency does not increase in proportion with threads

The preceding discussion on threads could easily bring in the notion that the more threads you have, the better. But it's not true. More threads run parallelly when run on multi-CPU systems, but if there are shared resources, the parallelism is lost due to resources (code and data) that act as bottlenecks.

This can be exemplified with a real-world case study. A thread-pool based Java application server hosting a Java web application showed the following data when profiled for CPU efficiency. The following table shows an exemplary (not an industrial benchmark, only a reasonable average) workload efficiency with a multi-threaded application:

Attributes	Value
Number of threads	10
Average CPU consumption	8%
Total workload efficiency	80%

Table 9.2: *Average workload efficiency in multi-threaded applications*

The result differs when the same workload is hosted on a single-thread Node.js runtime. The following table shows an exemplary (again, not an industrial benchmark, only a reasonable average) workload efficiency with a single-threaded Node.js application.

Attributes	Value
Number of threads	1
Average CPU consumption	93%
Total workload efficiency	93%

Table 9.3: *Average workload efficiency in single-threaded applications*

Evidently, a single thread can bring more workload efficiency than 10 threads. This is because the threads performing blocking I/O are idle for most of the time while holding on to resources. On the other hand, the single-threaded system performs I/O as non-blocking, and the wait time is effectively utilized for running other work (potentially the next request in the queue), increasing the overall efficiency.

Secondly, multiple threads introduce the burden of locking. With a single thread, the program data is automatically protected from cluttered access and devoid of the overhead of synchronization efforts and the wait time thereon.

In summary, the number of threads does not truly reflect efficiency/inefficiency. Instead, the amount of CPU utilization as a function of the available CPU count is a reasonably good measure of efficiency.

So, carefully assess the above-mentioned factors and determine whether scalability is a need before tuning your application for scalability improvements.

Issue

My system is underutilized despite the consideration of high CPU density through concurrency, as only a very small percentage (< 10%) of the CPUs are actually contributing to handling the load.

Remediation

There are at least three different architectural models that help a Node.js process to vertically scale and take advantage of the multiple CPUs in the system. Each model addresses the case differently and with different tradeoffs.

Multiple Node.js processes or child process

In this model, identical application replicas are started in the system to use the free CPUs and share the workload.

The following code snippet shows how to prepare the application to run its multiple instances in the same system (by parameterizing the port number):

```
1. const h = require('http')
2. const s = h.createServer((q, r) => {
3.   r.end('hello')
4. })
5.
6. s.listen(+process.argv[2])
```

And the following code snippet shows how to run the application's multiple instances in the same system (by assigning different port numbers to different instances):

```
1. $ node app.js 12000
2. $ node app.js 13000
3. $ node app.js 14000
4. $ node app.js 15000
5. $ node app.js 16000
```

Pros

- Each process is truly independent from the others, so it improves vertical scalability and availability in case one or more processes crash.

Cons

- Each process is running on the same system and one system can have one unique port, so multiple servers cannot listen at the same port at the same time. On the other hand, we need to expose a single port number to the external world.

- Also, if we resolved the unique port problem, how do we decide which server instance should handle which request? How do we ensure that the client requests are evenly distributed? So, there is the additional work of intercepting the request through a published port and routing requests to the different instances of the application.

These can be addressed by installing a load balancer (reverse proxy) in the middle, which receives the request and routes it to the actual Node.js processes in the system.

Cluster (Node.js module)

Don't confuse the cluster module with the general concept of cluster in distributed computing. While the objective is the same, we are discussing the Node.js module called **Cluster** here.

This module implements APIs specifically to address the two issues mentioned in the child process case:

- Unique port
- Balancing the load

The following code snippet shows how to run multiple instances of the application in the same system and share a single port number (using the Cluster API):

```
1. const c = require('cluster')
2. const h = require('http')
3.
4. if (c.isMaster) {
5.   c.fork()
6.   c.fork()
7.   c.fork()
8.   c.fork()
9.   c.fork()
10.  c.on('exit', (w, e, s) => {
11.    c.fork()
```

```
12.   })
13. } else {
14.   const s = h.createServer((q, r) => {
15.     res.end('hello')
16.   })
17.   s.listen(12000)
18. }
```

In the preceding code, we implement a web server with the help of a master and five workers. We create the workers in the master. Upon the exit of any worker, we create another worker to ensure that we always have five healthy workers. The 'fork' API of the **`cluster`** module is used to create a new worker process, and we set up a server instance in each worker. The key aspect to note here is the usage of the port. Although it appears that every worker process seems to be listening to port 12000, under the cover only one entity (the master) is actually listening, while the request is routed to any of the workers based on a predefined algorithm.

The following screenshot shows proof that all the cluster instances share a single port number and that the server is listening to accept incoming connections:

```
01. const h = require('http')
02. const s = h.createServer((q, r) => {
03.   r.end('hello')
04. })
05. s.listen(+process.argv[2])
```

Figure 9.14: *Proof for port sharing by all cluster members*

In this approach, there is one master process and several worker processes. In the most common model, the master process accepts the request and delegates further processing of the request to one of the worker nodes. The assignment happens in a round robin manner.

Pros

- With this approach, the multi-core leverage and load balancing are taken care of.
- Worker crashes do not affect the cluster. New workers can be spawned transparent to the frontend/client.

Cons

- Sessions are not managed by the module; that is a responsibility of the application itself.
- Crash to the master will destabilize all the workers, so there is a single point of criticality.

Worker threads

The worker threads module provides API support for first class threads in Node.js. These threads can run parallelly with the main thread and execute any arbitrary piece of code. In the most common use case, worker threads are used to run CPU-intensive operations that are off-loaded from the main thread. Web request sharing through workers is a natural expectation, but we haven't seen workers used in that manner yet.

The following code snippet shows how to run the web application with multiple threads in the same process, offloading CPU-intensive work to those threads (using the worker-threads API):

```
1. const worker = require('worker_threads')
2. const w = new worker.Worker('./worker.js')
3. w.on('message', (m) => {
4.   console.log(`worker message: ${m}`)
5. })
6. w.on('error', (e) => {
7.   console.log(`worker error: ${e}`)
8. })
9. w.on('exit', (c) => {
10.   console.log(`worker exit code: ${c}`)
11. })
```

In the preceding code, we create a worker thread and pass the script that it needs to run as an argument. In the parent, we install callbacks for message, potential error, or exit from the worker thread and handle those scenarios appropriately.

Pros

- Managing the worker thread life cycle is easiest among the three vertical scalability approaches: Child Process, Cluster, and Worker Threads (creation, launching, executing, data transport, and destroying)

Cons

- Sharing web workload is not easy

In summary, these three modules provide different capabilities to exploit multi-core and increase vertical scalability at different levels of API abstraction. One or a combination of these can be employed based on your specific use case.

Horizontal scalability

The best practice around horizontal scalability is to employ redundant copies of the same application (called **replicas**) in different systems and install a central dispatching entity (called **load balancer**) that controls the traffic between the replicas.

The following screenshot shows stacked up CPUs in multiple systems to represent the approach of horizontal scalability:

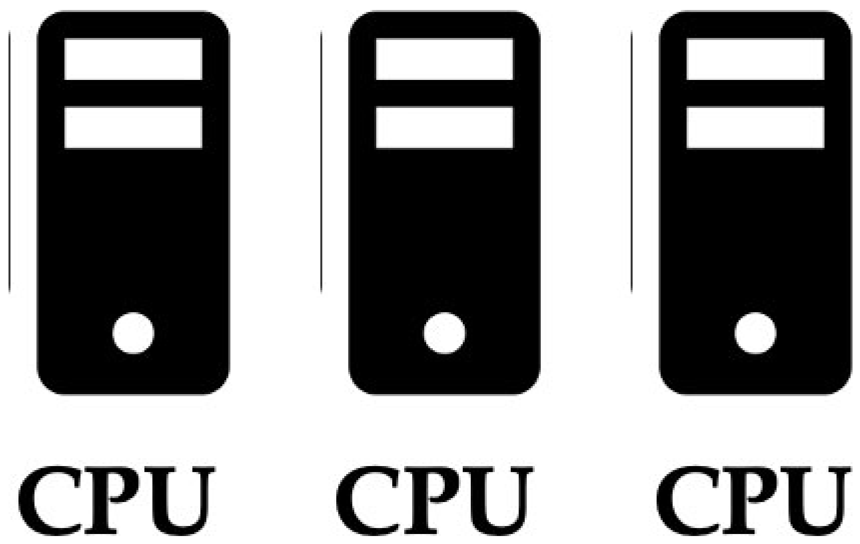

Figure 9.15: *Horizontal scaling example*

Auto-scale architecture is an application architecture in which fully independent replicas of application modules are deployed and used in response to an increase in the application's demand. First, the application itself is decoupled and decomposed into multiple services, and then each of those services is allowed to function independently. This produces optimal scalability outcome, though at the expense of a little extra allocation of resources per replica.

Observability

Observability is defined as the measure of visibility to an application's internal state, from the observable data that it produces. In other words, it is the degree of transparency the application's functions are exposed to the outside world, through the logging data it produces.

Bringing the definition to the context of a web application, observability is the ability of a web application to capture data of at least three levels of abstraction:

- **Application topology view**: The ability to visualize the shape of the application
- **Performance view**: The ability to visualize the performance characteristics
- **Individual transaction view**: The ability to dissect to each request-response cycle

In addition to this, we should capture data pertinent to anomalies (incidents) that occur in the application. The data can be a crash dump, a CPU spike graph, or a call drop trace. The data thus captured should be sufficient to figure out what happened to the application and why, when, and how it happened. The best practices listed here focus on these scenarios and ensure that the application is observable in a comprehensive manner, under all its stated use cases.

The following figure shows the key elements that constitute **Observability**:

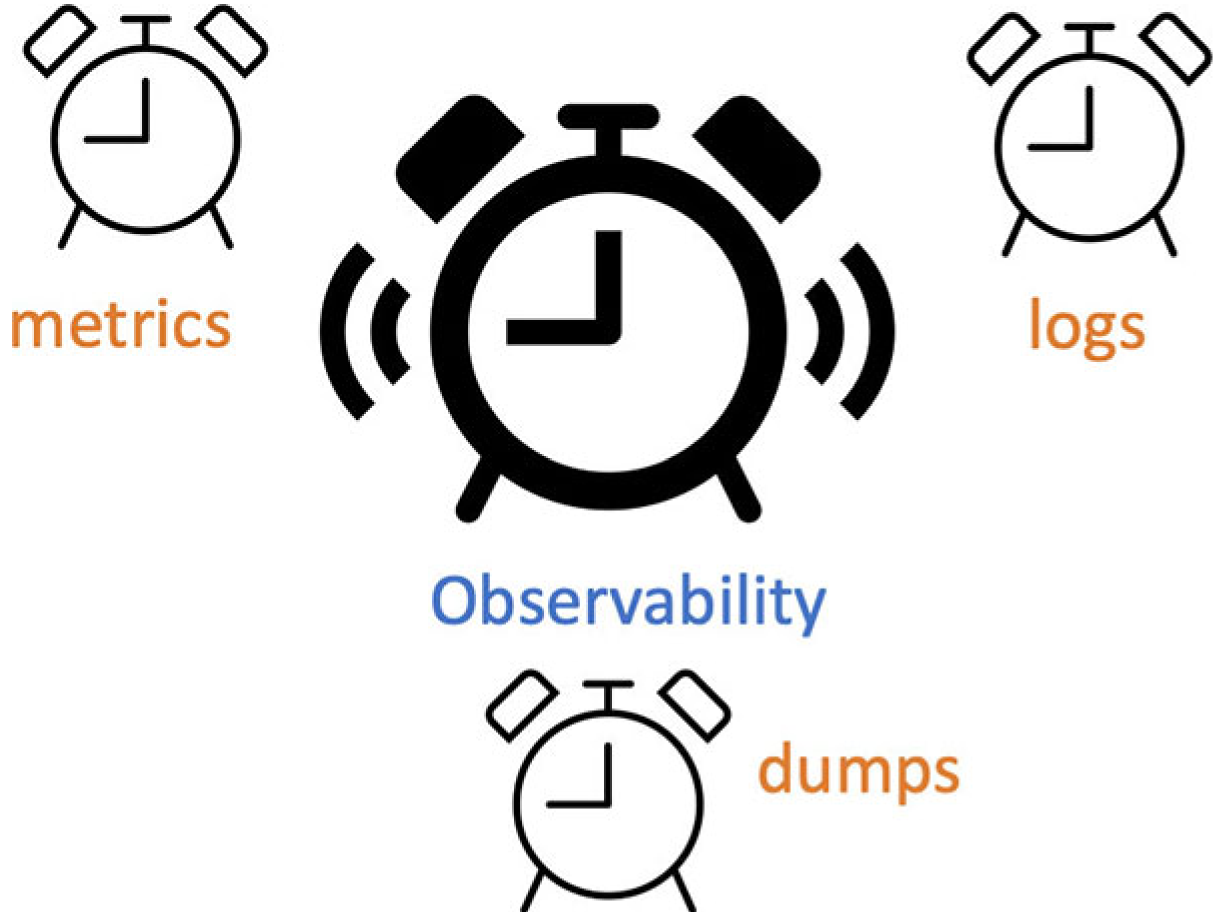

Figure 9.16: *Essential ingredients for observability*

To achieve all these, the comprehensive data capture will follow this form:

- Trace/log
- Profile/monitor
- Dump/snapshot

Trace (also known as log)

Trace is defined as the process of recording the information about a software execution. More specifically, it is the capturing of important information at vital control flow points in an application to visualize the sequence of actions (instructions/tasks) that were carried out in the program. Tracing and logging are used to represent different things in many technologies, but they are one and the same at the level of generalization we are describing.

The following screenshot shows an exemplary trace with the built-in `NODE_DEBUG` option enabled, which traces the specified module:

```
#export NODE_DEBUG=worker
#node w
WORKER 35051: [0] create new worker ./worker.js {}
WORKER 35051: [0] created Worker with ID 1
WORKER 35051: [0] received message MessageEvent {
```

Figure 9.17: *Example of executing application with Node.js debug tracing*

The following screenshot shows how to enable built-in trace for a Node.js application:

```
01.  $ node --trace app.js
```

Figure 9.18: *Command line to execute application with v8 tracing*

The following screenshot shows an exemplary trace with the built-in `--trace` option enabled, which traces the entire application:

```
01.     4: ~get+1(this=0xffdc <WriteStream map = 0xffef>) {
02.     4: } -> 0xffab <false>
```

Figure 9.19: *Example of a Node.js trace data*

Tracing is useful when we are inspecting a specific sequence of task or a single transaction. Here's a common debugging algorithm:

1. Line up the trace data for the entire span
2. Identify the earliest trace record with no anomaly
3. For each trace record, inspect the corresponding code in the source

4. Follow the control flow and the data flow
5. Repeat the last two steps until you catch the error

Some common practices around tracing are as follows:

- Ensure that the tracing frequency is normalized across the application, so try to maintain the same rate if you have a trace record per 100 lines of code in a module.
- Ensure that the tracing follows a hierarchy, that is, a trace record tells us which module that trace belongs to. For example, application logs, database logs, and network logs are discretely distinguishable from one another.
- Ensure that the trace record has a well-defined structure and is machine-readable so that it can also be fed to sophisticated visualization programs in addition to direct human consumption. Ensure that each record has a unique ID, a timestamp, and a clear message that reflects a specific decision point in the control flow and its associated data.
- Ensure that the trace is configurable and with different levels of intensity. This means a more intense tracing will produce maximum trace records, while a less intensive tracing will produce minimal trace data. As tracing incurs CPU time, there is always a tradeoff between traceability and performance.

Profiling/monitoring

Profiling is defined as the process of recording information about a software execution. More specifically, it refers to capturing important information about resource consumption in an application to visualize the performance characteristics.

Profiling is useful when we are inspecting a general sequence of task with the expectation of drilling down to a specific section of the code. Here's a common debugging algorithm:

1. Line up the profile data for the entire span
2. Identify the top space (memory)/time (CPU) consumer(s)
3. Narrow the profile data to cover the span of the top consumer

4. Identify the top consumers(s) again
5. Repeat the last two steps until you catch the hotspot

Ensure that the profiler can show hierarchical views in terms of top CPU (or memory) consumer:

- Modules, classes, functions, blocks, lines, instructions – for CPU
- Modules, libraries, classes, objects – for memory
- Application, middleware, frameworks, tools, runtime, OS – for both

The following screenshot shows an exemplary profile captured through a monitoring tool that shows the CPU utilization profile for various methods that took part in a transaction:

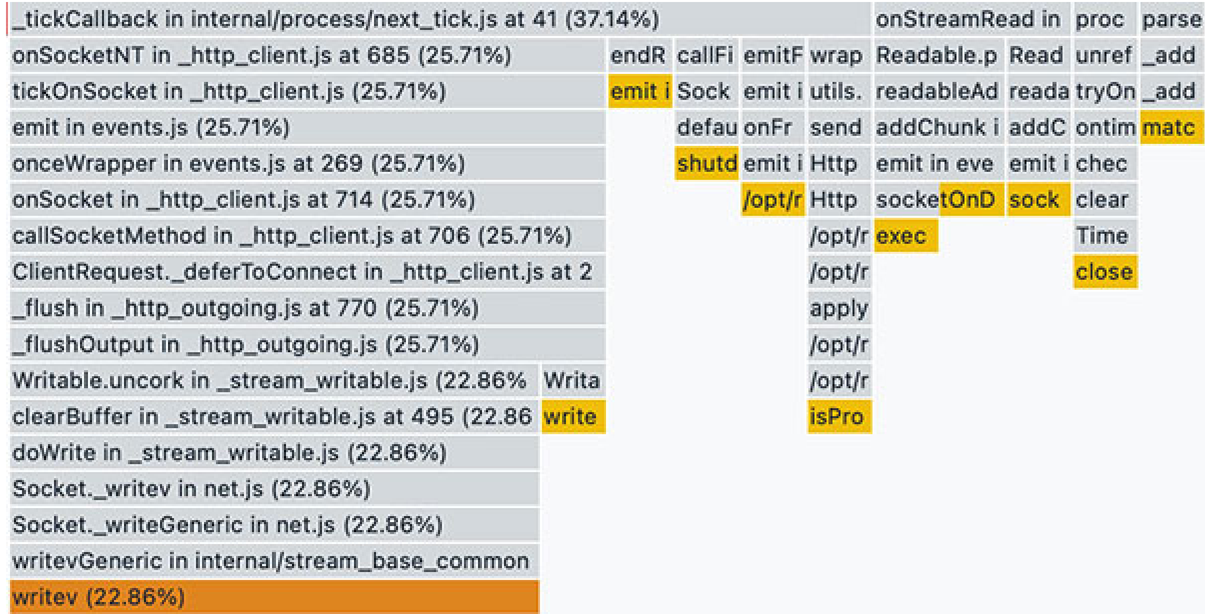

Figure 9.20: *Example of a CPU profile (aka flame graph)*

This hierarchy helps different people diagnose the bottlenecks at differing levels of abstraction. For example, an architect may want to see the most memory-savvy module with an objective of restructuring it, while a developer may want to see a set of instructions that is responsible for a specific latency value.

Similarly, the division of resources between various software stack components helps isolate the bottleneck in a specific component.

One of the important things about performance profiling is defining a baseline. This is carried out by running a desired workload and measuring and recording all the relevant parameters. Later, this is used for comparison for easy detection of outlier patterns and abnormal behavior.

Dumping/snapshotting

Dumping is defined as the process of recording information about a software execution. More specifically, it refers to capturing important information about the state of the application as an operating system process to visualize the internal state.

Generally, there are three types of dumps:

- System dump (aka core dump, minidump, core file, abend dump, core)
- Heap dump (aka heap snapshot, heap profile)
- Textual dump (aka snapshot)

A system dump contains the entire virtual address space of the running process and the internal state of the process; for example, the call-stack and register content of each thread, shared libraries, open file descriptors, and so on. We need a special debugger tool to launch and process the system dumps. Node.js dumps can be launched in native debuggers like gdb, lldb, windbg, and dbx. Some of the internal data structures are not understood by the native debuggers due to the managed runtime nature, so a debugger plugin like `llnode` comes in handy to have a sophisticated debug experience.

The following screenshot shows an exemplary stack trace captured through a debugging tool (gdb) that shows the function calling sequence within a core dump collected from a Node.js application:

```
01.  $ gdb node core
02.  (gdb) where
03.  #0  0x1c44062 in v8::base::OS::Abort() ()
04.  #1  0x0e83a81 in Isolate::CreateMessageOrAbort(Handle, MessageLocation*)
05.  #2  0x0e83bba in Isolate::ThrowInternal(Object, MessageLocation*)
06.  #3  0x0f83c0a in IC::ReferenceError(Handle)
07.  #4  0x0f8bc83 in LoadIC::Load(Handle, Handle, bool, Handle)
08.  #5  0x0f8c405 in Runtime_LoadNoFeedbackIC_Miss(int, unsigned long*, Isolate*)
09.  #6  0x1638af9 in Builtins_CEntry_Return1_DontSaveFPRegs_ArgvOnStack_NoBuiltinExit
10.  #7  0x16b5ab2 in Builtins_LdaGlobalHandler
11.  #13 0x1638af9 in Builtins_CEntry_Return1_DontSaveFPRegs_ArgvOnStack_NoBuiltinExit
12.  #14 0x15d1eab in Builtins_InterpreterEntryTrampoline
13.  (gdb)
```

Figure 9.21: Example of a stack trace from a system dump

A heap dump contains the entire content of the managed heap (aka object heap). We need special tools to launch and process the heap dumps. We can produce heap dumps at will using v8 APIs, or they are automatically generated when the heaps get exhausted.

The following screenshot shows an exemplary JavaScript heap captured through a debugging tool (heapdump) that shows the objects within the heap with their resident size:

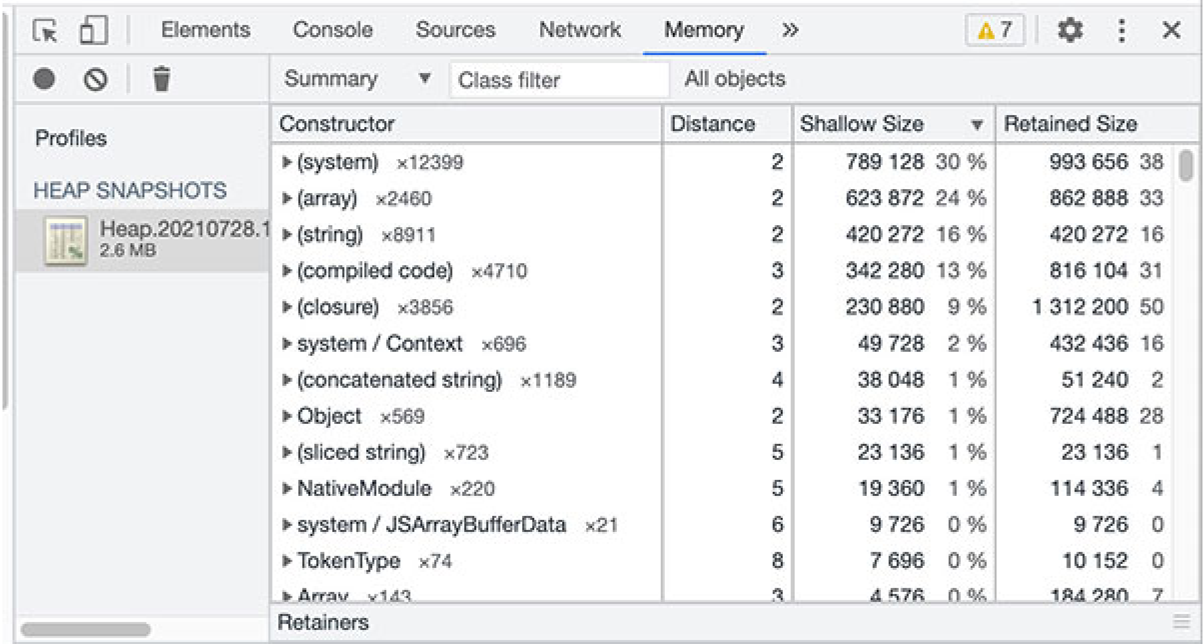

Figure 9.22: Example of a JavaScript heap snapshot

A textual dump contains an array of usual information about the application. This can be thought of as a combination of both process-level and application-level data that is easily consumable. For Node.js, diagnostic report is the text dump. It is produced on demand and automatically produced in the case of abnormal events.

The following screenshot shows an exemplary text dump captured through a debugging tool (diagnostic report) that shows the various internal states of the running process:

```
01. {
02.   "header": {
03.     "reportVersion": 2,
04.     "event": "y is not defined",
05.     "trigger": "Exception",
06.     "filename": "report.20210728.120900.36794.0.001.json",
07.     "dumpEventTime": "2021-07-28T12:09:00Z",
08.     "dumpEventTimeStamp": "1627454340539",
09.     "processId": 36794,
10.     "threadId": 0,
11.     "cwd": "/usr/gireesh/project",
12.     "commandLine": [
13.       "node",
14.       "--report-uncaught-exception",
15.       "core.js"
16.     ],
17.     "nodejsVersion": "v14.0.0",
18.     "wordSize": 64,
19.   }
20. }
```

Figure 9.23: *Example of a diagnostic report snapshot*

Dumping is useful when we are debugging issues at the lowest level with the ability to drill down to a specific section of the code and date. Here's a common debugging algorithm when debugging a system core file:

- Identify the problem context: code and data
- Locate the failing context in the dump
- Identify the call sequence that led to the failing context
- Walk backward in the call sequence
- Record the data transformation/code logic in the backward route
- Repeat the last two steps until you catch the bad code/data

In summary, tracing, profiling, and dumping are powerful techniques to know the code flow/data flow, performance, and the internal state of the application, respectively. It is an anti-pattern to use these techniques interchangeably; for example, using tracing to figure out the state of the application, dumping to detect the application flow, and so on. While they give you some result, they are neither comprehensive nor optimal.

Security

Security is defined as the measures taken in the software to protect it against intended or unintended abuse leading to loss of data, business, and reputation. In web application's context, security is of paramount relevance and importance as most software abuse targets web applications.

We have discussed the driving principle of web application security in *Chapter 3, Introduction to Web Server*. Since the application resides in a remote host and is accessed through the internet, the software becomes subject to illegal access and potential manipulation. So, securing a web application means:

- Ensuring that only valid users are accessing the service
- Ensuring that access is restricted to specific areas for valid users

All the security best practices are fundamentally one or the other form of these.

Input validation

Given that the server processing is largely driven by input (user request), a class of security threats use crafted input to drive the web application into vulnerable control flow paths. So, the input should be thoroughly sanitized even before the request processing begins.

The following screenshot shows an exemplary request URL that shows an embedded query that is potentially vulnerable if processed as is (or un-sanitized):

```
01. https://www.mywebsite.com/account?query=select%20%2Afrom%20accouont_registry
```

Figure 9.24: *Example of a vulnerable request header*

This can be direct, such as an SQL query in the user input driving the server to execute it as is, or indirect, such as a number in the query string driving the server to allocate as many bytes of memory to hold a string. In either case, care must be taken to validate the request and assert that it is genuine and will not break the server.

Other examples where user input can mislead server actions are regular expressions, evaluation function, timer intervals, child process strings,

operating system commands, and so on.

Secure headers

Use security headers like X-content-type-options, X-frame-options, and content-security-policy to protect against cross-site scripting and click-hijacking.

The following screenshot shows a secure request header:

```
01.  Cache-Control: private, no-cache, no-store, must-revalidate
02.  X-XSS-Protection: 1; mode=block
03.  X-Frame-Options: SAMEORIGIN
```

Figure 9.25: *Example of a secure request header*

Secure sessions

Create strong, unique session identifiers with proven algorithms. Ensure that the sessions are well confined to the scope of the user's login session, and ensure that the session has a reasonable life span. Too short an interval can affect user experience, while too long an interval can lead to security issues.

The following code snippet shows an exemplary secure session cookie object creation:

```
1. const session = new Session({
2.   secret: 'top secret',
3.   resave: false,
4.   saveUninitialized: false,
5.   cookie: {
6.     secure: true,
7.     maxAge: 20000
8.   }
9. })
```

Authentication and authorization

In simple words, we must institute controls to ensure that only valid users can enter the service and only a well-defined part of the service is accessible to a group of users. This is called role-based access control.

The following code snippet showcases authentication and authorization:

```
1. function authenticate(user, ctx) {
2.   if (!user.cred.matches(ctx.cred))
3.     reject()
4. }
5.
6. function authorize(user, ctx)
7. if (!user.role(matches(ctx.access))
8.   reject()
9. }
```

Data encryption

Ensure that valuable business data that flows between the client and server is always encrypted. Though this does not directly impact the server's functioning, the data that gets stolen if sent unencrypted can lead to credibility loss for your web application. So, always use secure protocol for data transport.

The following code snippet demonstrates secure data transport:

```
1. const crypto = require('crypto')
2. const data = 'top secret'
3. const buffer = Buffer.from(data)
4. crypto.publicEncrypt(buffer, rsa_pubenc)
```

Audit logging (aka logging for security)

Security incidents like executing arbitrary code and taking undesired control flow can be detected later if there is adequate logging in your application. So, apart from the logging that aids observability, ensure that the audit log is taken care of in the application at the design phase itself.

The following code snippet shows logging for security (audit log):

```
01. incident: {
02.   timestamp: 1627470374750,
03.   request: 'approve',
04.   requester: '56gc-opq8',
05.   reason_code: '800Q',
06.   response: 'OK',
07.   id: 'dsdhb12-sdsykjdwd-878dsj'
08. }
```

Figure 9.26: *Example of a security log*

File system protection

Ensure that the web application has well-defined access to the file system. The application code and the configuration are maintained as read-only for the user who runs the application. That way, the process cannot corrupt the code and configuration in the case of an exploit.

The following code snippet shows a secure file system management of server resources:

```
01.  -r-r--r--  1 nodeuser  staff  256520 Jul 28 16:28 app.js
02. drwxr-xr-x  2 nodeuser  staff      64 Jul 28 16:28 public
03.  -r-r--r--  1 nodeuser  staff  434320 Jul 28 16:28 lib.js
04.  -r-r--r--  1 nodeuser  staff   24234 Jul 28 16:28 config.json
```

Figure 9.27: *Example that shows how to secure server file system*

Shutdown unwanted ports

Use networking primitives to close unused ports in the system as open and lingering ports allow attackers to use them to send content to the outside world if the system is compromised. Also, open ports allow attackers to gain better information about your network, operating system, and such without breaking into the system.

Documentation

Documentation is a text that explains the software's capabilities and usage. In the context of a web application, it is composed of at least three things:

- Topology of the website (or a site map)
- Feature documentation (or user guide)
- Troubleshooting guide (or support document)

The philosophy of web application documentation is to be able to follow the documentation to use the web application in its entirety, with minimum or zero manual intervention and support from the vendor. Ideally, users do not want to read through long documents to understand how a web application works, but from the application's perspective, it is great to have a compact document. Some simple best practices are:

- Categorize the content. Ensure that the overall content is organized based on the high-level features that the application is offering.
- Show screenshots when describing the interactive view part of the feature. That way, the consumption becomes faster and atomic.
- Give examples wherever possible as opposed to explaining generically or in an abstract manner. Examples are easily consumed as opposed to definitions.
- FAQ-style content is highly consumable too.
- Inline help in forms. Some fields are difficult to fill in; for example, "domain" - the user may not be able to fill it as it is hard to understand. Some fields (like passwords) may have complex content rules. So, a tooltip that explains the field context when the user hovers over it will be very useful.
- Make the document or document link available on every page. That way, user can access it wherever they are.
- Document the contact information. A web application with contact information is a trustworthy site as opposed to a site without one.

Conclusion

In this chapter, we examined non-functional factors that are important in terms of hosting an industrial-strength and enterprise grade website, like reliability, availability, scalability, security, observability, and documentation. We looked at several best practices to strengthen our application against each of these attributes. We gathered some generic knowledge that is relevant for any software and specific know-how for web applications, plus we looked at more concrete scenarios for a Node.js-based web application. One of the key attributes that we omitted here is performance, and we want to look at it a little more. So, the next chapter is dedicated to performance best practices in a web application context, touching on the software stack involved.

CHAPTER 10
Best Practices for High Performant Code

In the previous chapter, we examined non-functional factors that are important in terms of hosting an industrial-strength, enterprise-grade web application. We studied best practices that strengthen our application toward each of these attributes. One of the key attributes that we omitted there was performance, and we want to expand that a little more and examine it exclusively. So, we are focusing on the performance aspect of our web application in this chapter, fine tuning all the software stack involved. This will help us understand the execution environment of our application at its finest level of details, get an insight into various tradeoffs that play roles in the application's performance aspects, and apply best practices that are relevant and known in the area.

Structure

We will cover the following topics in this chapter:

- Performance best practice: Hardware
- Performance best practice: Network
- Performance best practice: Operating system
- Performance best practice: Runtime (Node.js)
- Performance best practice: Application

Objective

After studying this chapter, you will be able to understand the performance characteristics of a web application and various best practices to improve its performance from the ground up. You will learn about the specific considerations for the selection of hardware, operating system, and network, and you will understand the specific configurations for the runtime platform

and the application. You will be able to follow these practices independent of the application's deployment model. Many of these best practices are generic and useful for any application, some are for web applications, while some others are specific for Node.js-based applications due to their peculiar characteristics and behavior on web workloads.

Performance best practice: Hardware

The server runs instructions on the underlying hardware, so the systems must be robust (resilient to failures) and powerful (able to execute several millions of instructions in unit time). It should be able to run for months without needing to restart. Let's look at the specific aspects of the hardware that are relevant to the performance of a Node.js web application.

CPU

In the previous chapter, we described the multi-core exploitation patterns of a Node.js application in detail. The number of CPUs may be selected based on your specific case. For example, if you are using an in-built Cluster module with 10 worker nodes, each worker node will have one Node.js process, and each will have 11 threads, making up a total of 110 threads. A 120 CPU system will be a reasonable match for such an application.

The preceding calculation is a rough estimate but a practical approach to computing the CPU count. For example, not all 11 threads will need active CPU throughout the application runtime. Except the main thread (the application thread), all other background helper threads will typically consume much less CPU, so the application in the above example can run well even with 70-80 CPUs. But these approximations are reasonable to make given that every workload will have a certain level of variation either way for CPU demand.

Installing a lot of spare CPUs does not add value to the application as they are unutilized and wasted. On the other hand, having less CPUs than required causes CPU starvation to the threads and will lead to poor performance. So, the key is to count the number of threads in the application and design a system that matches the count.

The bottom line here is to allow the Node.js process to run on enough number of CPUs so that each process thread can run on a separate CPU.

Note: If you are hosting your application on a container orchestration system, the load balancing is performed by the orchestration system in a horizontal manner—auto scaling with the help of new replicas of the application. In that scenario, there is no need for vertical scalability and a large number of CPUs in the hardware.

Cache

CPU cache are memory devices that are volatile memory for storing in-flight code and data for programs. CPU caches are faster than main memory (RAM) but slower than the CPU register, so frequently used data can be stored in cache instead of the main memory (that needs address translation, optional memory bus locking, and so on). L1 cache are the fastest but available in KBs, followed by L2 and L3 caches.

The following screenshot shows the contrast between performance and space of various internal storage devices:

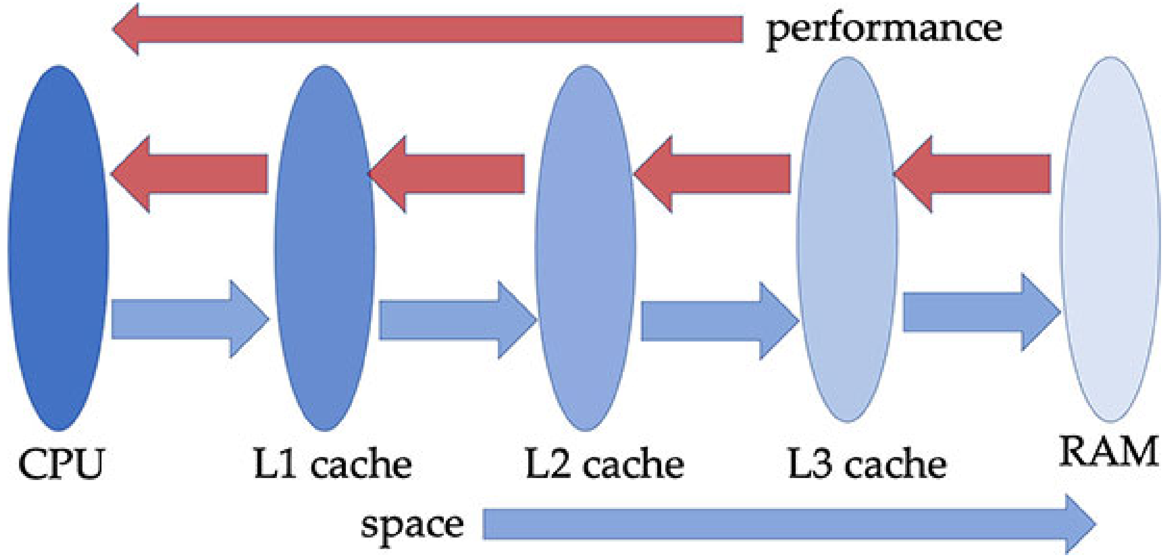

Figure 10.1: *Space-time tradeoff between various data storage devices*

The usage of cache improves the overall application performance, but it is not reasonable to expect these caches to hold all the program code and data of an enterprise application, even if we select the most frequently used objects as cacheable candidates. The reason is that a large application will have too much data to be held in cache. So, at some point in time, useful data gets evicted into main memory, leading to cache miss. Cache miss is not a program error, but it affects performance.

Cache miss is more pronounced for applications that run managed runtimes than for those running applications developed in native languages. This is because there are two or more discrete layers in the application stack with respect to data, and their locality of reference is distorted. For example, an application that deals with the JavaScript heap is in one specific address range, and the data that the virtual machine deals with for its business logic is in another specific address range. As a result, caches are a little less efficient in managed runtimes, though JIT compilers can rectify this to some extent.

The cache miss is even more pronounced in Node.js due to the high level of concurrency. The same thread is multiplexing between many transactions, so the data used for Nth transaction may be useless for (N+1)th transaction that comes for execution and may evict the cache content to put its own data, but the Nth transaction will find its own data out of cache when it resumes its execution.

A best practice is to use large L2 cache for your system. In other words, keep L2 cache size as a key factor when you are selecting your hardware.

Disable **Symmetric Multi-threading** (**SMT**) in the system. Enough physical CPUs to run the workload will help schedule each thread on a physical CPU, and logical CPUs with SMT may prove counterproductive as SMT threads share CPU cache and can pollute each other.

Note: Unfortunately, there is no easy way to measure and validate cache efficiency and map it to the granularity of program objects. However, there are operating system commands that show cache hit and cache miss on a per-process basis; these can be used to iteratively understand the cache characteristics of your process.

Disk

If your application is not a file server or a file upload destination, you do not really need to worry about the disk specification. This is because the code and configuration required to run the application are loaded into the memory before use and remain there for the life of the application. So, the only thing to take care of is to ensure that the disk has ample storage space for the application, any logs it may produce, and potential diagnostic artifacts like heap dumps or snapshots.

Performance best practice: Network

Network tuning plays an important role in the performance characteristics of Node.js web applications. However, the precise configuration will vary from one platform to another and one application to another.

Keeping performance as the paramount factor, a general best practice is to inspect the network tuning parameters, review their implications on your specific application, and adjust the network parameters accordingly. Specific and limited example configurations are given here:

It is a good practice to enable the TCP window scaling for applications that deal with large data, for sending or receiving. In Linux, this can be done as follows:

```
1. $ sudo sysctl -w net.ipv4.tcp_window_scaling=1
```

For the same type of applications, it is a good practice to modify the operating system's send and receive buffer sizes. An example setting is as follows:

```
1. $ sudo sysctl -w net.core.rmem_max=16777216
2. $ sudo sysctl -w net.core.wmem_max=16777216
```

The individual socket buffer sizes can also be modified as follows:

```
1. $ sudo sysctl -w net.ipv4.tcp_rmem="4096 87380 16777216"
2. $ sudo sysctl -w net.ipv4.tcp_wmem="4096 16384 16777216"
```

As a rule of thumb, set the network parameters based on what is permissible in your network and leverage that to the maximum.

Disabling Nagle's algorithm on the participating sockets is a good idea. This will ensure that the data is dispatched instantly, as and when it is submitted by the application to the TCP component, without internal buffering.

> **Note:** Nagle's algorithm addresses the "small packet problem" in the network data transport. Years ago, when the network bandwidth was slow and costly, data sent in terms of a small number of bytes caused unwanted congestion in the network. To remedy this, the algorithm suggests buffering the data internally and sending it only when an internal buffer has at least a predefined amount of bytes or a specific period of time has elapsed. This algorithm may prove counter-productive in modern day's highly concurrent workload running on high speed and relatively cheap network.

The `SetNoDelay` API is available on the socket as well as ClientRequest abstractions. This API is used to toggle the usage of Nagle's algorithm on the socket through which we are attempting to transport data.

Note: Network configuration parameters drastically change between operating systems, so we must carefully select specific options. Do not assume that a concept or command that is prevalent in an operating system carries over the same meaning in another one.

Performance best practice: Operating system

Ensure that the running application process has access to full breadth of the resources that are available in the system. Dedicate everything that the system has to the application's disposal so that the process is not constrained by lack of computing resources.

Ensure that the process gets enough memory to hold the data sections. This can be done as follows in UNIX systems:

```
1. $ ulimit -d unlimited
```

Ensure that the process can open the necessary file descriptors (files and sockets) as:

```
1. $ ulimit -n unlimited
```

Ensure that the process gets enough memory to hold the program data as:

```
1. $ ulimit -m unlimited
```

Ensure that the process gets enough CPU to execute tasks as:

```
1. $ ulimit -t unlimited
```

Ensure that the process gets enough virtual memory spectrum as:

```
1. $ ulimit -v unlimited
```

If there are no memory pages available to satisfy the current application's demand, the operating system will offload some part of the application's memory to secondary storage, which is considerably slower than the main memory. This will impact your application's performance. Ensure that the process memory does not get paged or swapped often. If possible, avoid the

use of disk swapping by installing an ample amount of real memory, preferably at least double than the peak memory usage of the application.

Ensure not to run other, unrelated process in the system. More processes add pressure to the system resources—scheduler, CPU, cache, disk, network device, and memory bus—and can adversely affect our application's performance.

Further, use the tools provided by the operating system to monitor the system under load and ensure that our configuration is ratified by the statistics. This will vary from one system to another. In a Linux system, these commands are typically useful:

1. The **`/proc`** file system provides live statistics of process, CPU, memory, disk, and network characteristics on a per-process and system-wide basis.

   The following snapshot shows the content of the virtual file system "proc" that essentially covers the dynamic attributes of all running processes:

```
01. $ ls /proc
02. 1      2    40  locks      loadavg
03. 10     20   41  interrupts ioports
04. 10068  128  21  meminfo    vmstat
05. 10073  13   22  modules    fs
06. 10393  1307 23  mounts     cgroups
07. 10394  1308 24  cmdline    net
08. 10463  132  25  consoles   cpuinfo
09. 10475  1311 26  devices    stat
10. 10476  1329 27  self       swaps
11. 114133 1337 34  sys        tty
```

Figure 10.2: *Sample /proc file system content*

2. The "top" command provides system-wide and per-process attributes of resource usage.

   The following snapshot shows the output of the "**`top`**" command, which provides a different view of running processes' resource usage:

```
01.  $top
02.  top - 10:10:29 up 2 days,  2:04,  2 users,  load average: 0.00, 0.00, 0.00
03.  Tasks: 185 total,   1 running, 184 sleeping,   0 stopped,   0 zombie
04.  %Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
05.  MiB Mem :  16012.9 total,  15118.8 free,    280.2 used,    613.8 buff/cache
06.  MiB Swap:  20480.0 total,  20375.7 free,    104.2 used.  15443.7 avail Mem
07.
08.      PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
09.     1183 root      20   0 1640140   8180   2672 S   0.3   0.0   0:57.15 contain+
10.    10661 nodeuser  20   0    9248   3860   3220 R   0.3   0.0   0:00.02 top
11.        1 root      20   0  170828  10372   6384 S   0.0   0.1   0:06.93 systemd
12.        2 root      20   0       0      0      0 S   0.0   0.0   0:00.06 kthreadd
13.        3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
14.        4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par+
15.        6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
16.        9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_perc+
17.       10 root      20   0       0      0      0 S   0.0   0.0   0:00.15 ksoftir+
18.       11 root      20   0       0      0      0 I   0.0   0.0   0:08.12 rcu_sch+
19.       12 root      rt   0       0      0      0 S   0.0   0.0   0:00.53 migrati+
20.       13 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_in+
21.       14 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
```

Figure 10.3: *Sample top command output*

3. "**`vmstat`**" provides system-wide attributes of processes, memory, paging, block IO, traps, disks, and CPU activity.

   The following screenshot shows the "**`vmstat`**" output that provides memory-specific usage statistics across the system:

```
01. $ vmstat -s
02.       16397212 K total memory
03.         282008 K used memory
04.         380640 K active memory
05.         224472 K inactive memory
06.       15495308 K free memory
07.          11148 K buffer memory
08.         608748 K swap cache
09.       20971512 K total swap
10.         106752 K used swap
11.       20864760 K free swap
12.         183567 non-nice user cpu ticks
13.           2085 nice user cpu ticks
14.          53340 system cpu ticks
15.      139570959 idle cpu ticks
16.          33723 IO-wait cpu ticks
17.              0 IRQ cpu ticks
18.           2991 softirq cpu ticks
19.            142 stolen cpu ticks
20.        7644661 pages paged in
21.        7078151 pages paged out
22.        1151816 pages swapped in
23.        1405627 pages swapped out
24.       17840333 interrupts
25.       26838474 CPU context switches
26.     1628521530 boot time
27.          10546 forks
```

Figure 10.4: *Sample vmstat command output*

4. "**`mpstat`**" provides system-wide attributes of the CPUs.

   The following snapshot shows the output of "mpstat", which displays various dynamic attributes of the available processors in the system:

```
01. $ mpstat -P ALL
02. 10:13:49 AM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
03. 10:13:49 AM  all    0.13    0.00    0.04    0.02    0.00    0.00    0.00    0.00    0.00   99.81
04. 10:13:49 AM    0    0.11    0.00    0.03    0.03    0.00    0.00    0.00    0.00    0.00   99.82
05. 10:13:49 AM    1    0.13    0.00    0.04    0.03    0.00    0.00    0.00    0.00    0.00   99.79
06. 10:13:49 AM    2    0.16    0.00    0.05    0.03    0.00    0.00    0.00    0.00    0.00   99.76
07. 10:13:49 AM    3    0.13    0.00    0.03    0.02    0.00    0.00    0.00    0.00    0.00   99.82
08. 10:13:49 AM    4    0.13    0.00    0.03    0.01    0.00    0.00    0.00    0.00    0.00   99.82
09. 10:13:49 AM    5    0.13    0.00    0.03    0.01    0.00    0.00    0.00    0.00    0.00   99.82
10. 10:13:49 AM    6    0.12    0.00    0.03    0.03    0.00    0.00    0.00    0.00    0.00   99.81
11. 10:13:49 AM    7    0.11    0.00    0.04    0.02    0.00    0.00    0.00    0.00    0.00   99.82
```

__Figure 10.5:__ Sample mpstat command output

5. "**`iostat`**" provides system-wide attributes of CPU and I/O devices.

   The following snapshot shows various "I/O" statistics in the system:

```
01. $ iostat
02. avg-cpu:  %user   %nice %system %iowait  %steal   %idle
03.            0.13    0.00    0.04    0.02    0.00   99.81
04.
05. Device             tps    kB_read/s    kB_wrtn/s    kB_dscd/s    kB_read    kB_wrtn    kB_dscd
06. dm-0              0.00         0.02         0.00         0.00       4308          0          0
07. dm-1             14.68        42.09        39.30         0.00    7604754    7100944          0
08. loop0             0.00         0.00         0.00         0.00        349          0          0
09. loop1             0.00         0.00         0.00         0.00        350          0          0
10. loop2             0.00         0.00         0.00         0.00        337          0          0
11. loop3             0.01         0.01         0.00         0.00       1829          0          0
12. loop4             0.00         0.01         0.00         0.00       1425          0          0
13. loop5             0.00         0.01         0.00         0.00       1089          0          0
14. loop6             0.00         0.00         0.00         0.00        336          0          0
15. loop7             0.19         0.19         0.00         0.00      33986          0          0
16. loop8             0.00         0.01         0.00         0.00       1066          0          0
17. vda               1.56        42.14        39.21         0.00    7613966    7083874          0
```

__Figure 10.6:__ Sample iostat command output

Note: The operating system that we run our application on is a general-purpose operating system designed for operating in the multi-user multi-tasking mode. However, we would want to undo or reverse some of those capabilities and defaults that were set for a general-purpose multi-tasking operating system when it is used for running an application exclusively and dedicatedly. Most of the above-mentioned best practices are precisely doing that.

Performance best practice: Runtime (Node.js)

In this section, we will look at the specific best practices that can be applied at the runtime level, that is, Node.js platform. We will look at the important components, like the garbage collector and the event loop, and learn how to fine-tune those components to yield better performance for our workload.

Garbage collection

Generational garbage collection policy perfectly complements the web workload. This is because the generational collection policy works on a principle that objects are short-lived and request-response based transactional workload is the most appropriate workload type for leveraging the best out of generational garbage collector. In such workloads, objects that got created as part of a transaction are generally eligible for being collected after the response

has been sent. There may be exceptions, such as audit data we want to store/persist beyond the transactions and global data we want to reuse across transactions, but those are generally limited as compared to the objects pertinent to the transactions.

The following screenshot gives a view of generational garbage collection model with the new and old heap spaces and the object residence pattern in those:

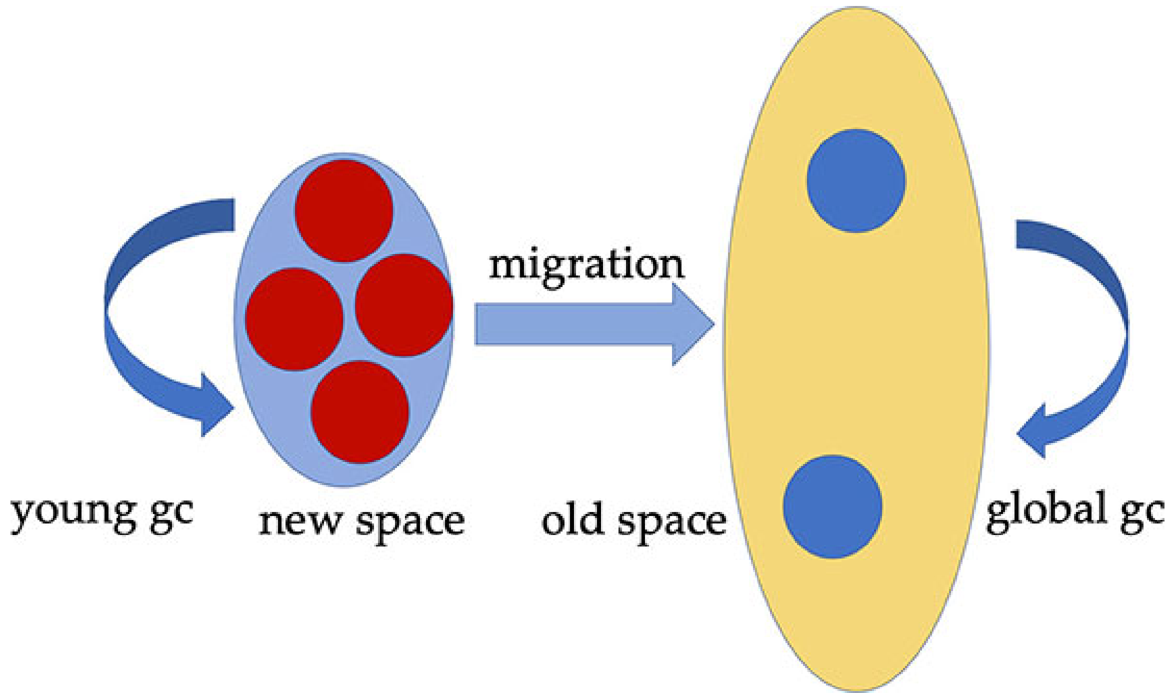

Figure 10.7: *Generational garbage collection architecture*

Carry out the following to leverage the best of the generational garbage collection policy that Node.js uses under the cover and improve performance:

1. Select a test workload characterized by your real-world use case
2. Match the concurrent user count of the test workload with the real-world use case
3. Select default new space and old space for the JavaScript heap
4. Run the workload with garbage collection tracing enabled
5. Record the frequency of new and old space collections
6. Repeat the test by increasing the default new and old spaces
7. Record the frequency of collections again
8. Repeat this process until:

- Maximum objects are collected in the new space itself
- Long intervals are realized between global collections

The following command illustrates how to modify the new space size of JavaScript heap:

```
1. $ node --max-semi-space-size=1024 app.js
```

The following command illustrates how to modify the old space size of JavaScript heap:

```
1. $ node --max-old-space-size=4096 app.js
```

The following command illustrates how to trace garbage collection in the process:

```
1. $ node --trace-gc app.js
```

The following screenshot illustrates the desired state of the heap spaces after the heap tuning exercise has been performed and the object allocation is optimized:

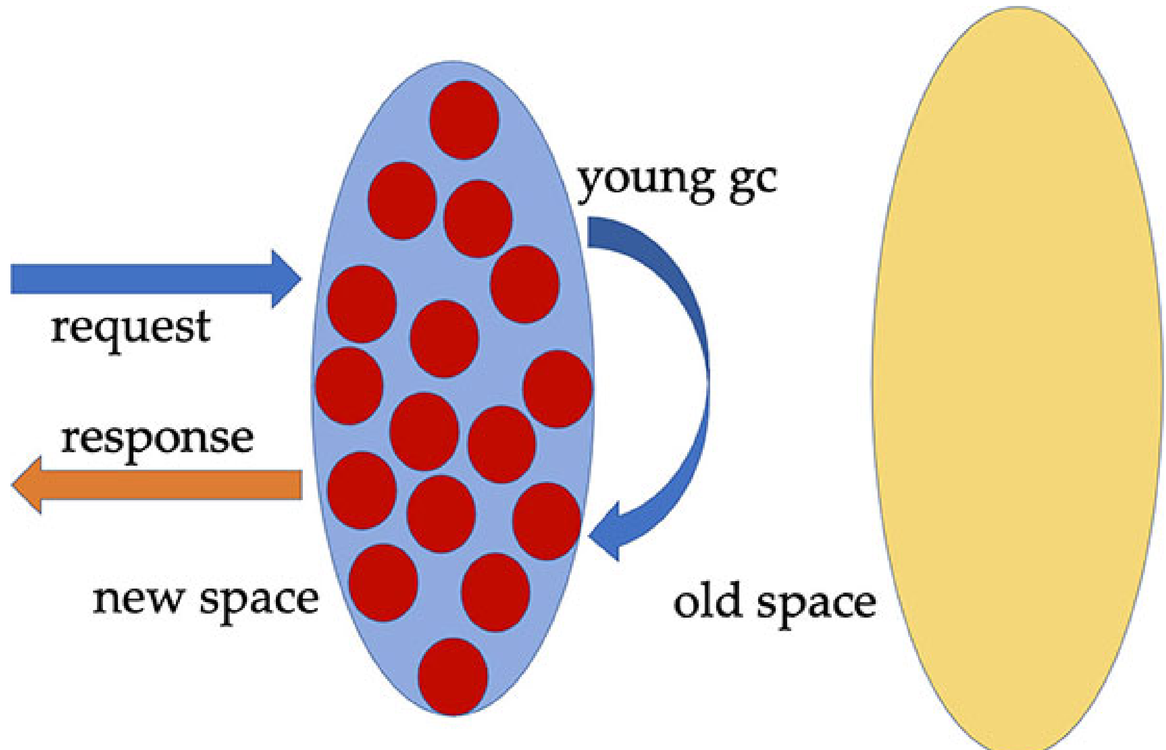

Figure 10.8: *Optimal generation collection state for web applications*

The bottom line of this algorithm is to avoid frequent global garbage collections by accommodating many transactional (request-response cycle) objects in the young space collections.

Avoid frequent garbage collections as it adds pressure on the application. Based on the specific characteristics of the application and the layout of objects in memory, the in-flight transactions can experience above-normal latency.

The following screenshot illustrates a misconfiguration, wherein the application is causing frequent garbage collection and resulting in turbulence to the application:

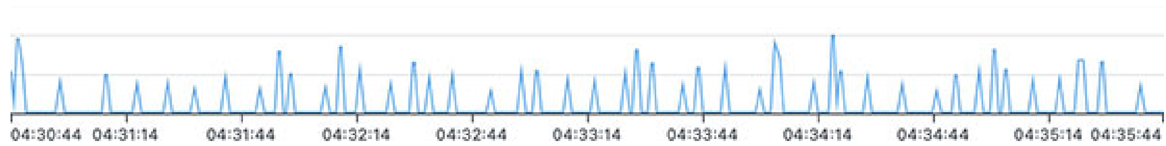

Figure 10.9: *Frequent gc pause times from a poorly configured JavaScript heap*

Avoid creating large objects. If the usage of large objects is inevitable, try to understand the premise in which those are created and see if it can be circumvented by application redesign; for example, splitting them into multiple objects.

Ensure that the JavaScript heap size is large enough to hold the peak time load. The peak time load can be measured using operating system tools like “top” in Linux. A thumb rule is to set the old space to 1.5 times the peak load demand.

Event loop

Event loop is an integral part of the Node.js architecture; in fact, it is the central part. This is because everything is executed as a response to an event after the initial warm-up and preparation of your web application server in event-driven architecture. So, the event interception and dispatch should occur on time—as soon as the event is originated and made available to the application—to obtain the required performance boost. In short, the event loop should be running at frequent intervals. Tight program loop constructs prove detrimental to this requirement as they block the progress of the single thread toward executing the event loop.

The following code snippet illustrates how performance can be degraded with loops in the code with arbitrary/unknown loop counter values:

```
1. const f = require('fs')
2.
3. function digest(file) {
4.   const d = f.readFile(file, (e, d) => {
5.     let l = 0
6.     for (var i = 0; i < d.length - 2; i++)
7.       l = (l << 8) | d[i + 2]
8.     return l
9.   })
10. }
```

In the preceding example, the "**digest**" function takes a file name as an input, reads it, applies some transformation to each byte, adds it up to a variable, and returns the value. The issue here is that the loop iteration count is equal to the number of bytes in the file; for example, the loop will run for 1024 * 1024 * 1024 times for a 1GB file.

Avoid tight "for" and "while" loops that run for more than 10 milliseconds.

How to avoid those? For example, a stable workaround is to do the following if our application's business logic demands it:

- Convert the code part that involves tight loops into a function
- Convert the function into an asynchronous call
- Create a worker thread and delegate it to execute that function
- Use the result from the worker thread asynchronously
- Continue the execution post the tight loop in the completion callback

The following code snippet illustrates how such tight loops with arbitrary loop counters can be converted to asynchronous executions with the help of worker threads:

```
1. const w = require('worker_threads')
2. const f = require('fs')
3.
4. if (w.isMainThread) {
5.   const worker = new w.Worker(__filename,
```

```
6.     {workerData: './worker.js'})
7.   worker.on('message', (m) => {
8.     console.log(m)
9.   })
10. } else {
11.   const file = w.workerData
12.   const d = f.readFile(file, (e, d) => {
13.     let l = 0
14.     for (var i = 0; i < d.length; i++)
15.       l = (l << 8) | d[i + 2]
16.     w.parentPort.postMessage(l)
17.   })
18. }
```

In lines 4 through 9 of the preceding example, a worker thread is created with the data filename as the input, and a message loop is put up to obtain the result from the worker thread. Lines 10 through 18 are executed in the worker thread, wherein the file is read and the loop code is executed. Given that this whole sequence is executed in a separate CPU, the loop count does not affect the main thread in any manner.

> **Note:** To reiterate the philosophy of event loop with a single thread: we (the single thread of execution) execute all CPU-bound operations inline, and when we encounter an I/O bound operation, we push it to the event loop and continue with the rest of the code. The pushed request will execute asynchronously in the system, return with the result, and continue execution in the designated callback.

Concurrency: "sync" versus "async"

Overall application performance is a function of the level of concurrency in a Node.js application, and the level of concurrency is a function of the density of asynchronous code executions. There are several synchronous APIs in the Node.js core, and they are provided for use cases where concurrency is not a concern. Avoid synchronous APIs as much as possible. If they need to be used in unavoidable circumstances and take more than 10 milliseconds to complete,

consider instituting a worker thread and run the synchronous API in there, while managing the communication between the parent and child threads through asynchronous messages.

There is an asynchronous counterpart for almost every synchronous API.

The following code snippet shows a synchronous version of a file copy API:

```
1. const f = require('fs')
2. f.copyFileSync('sync.js', 'async.js')
3. console.log('copy success')
```

And the following code snippet shows its asynchronous counterpart:

```
1. const f = require('fs')
2. f.copyFile('async.js', 'sync.js', (e) => {
3. if (!e)
4.   console.log('copy success')
5. })
```

Increasing concurrency

While we classified the slow computational operations as I/O-bound and fast ones as CPU-bound, several I/O operations cannot be performed asynchronously. Here are some examples of slow operations that are not supported by the operating system to be performed asynchronously:

- Disk I/O
- Cryptographic computations
- DNS lookup
- Compression and decompression

The asynchronous versions of these capabilities are simulated using internal helper threads that carry out tasks in a synchronous manner in-thread but act asynchronous with the application thread. The number of such threads is chosen at a convenient default but is controlled by an environment variable—**`UV_THREADPOOL_SIZE`**. While its default value of 4 is good for normal scenarios, adjusting this value based on the specific workload can make a difference. For example, an experiment in a Linux system with 16 logical

CPUs showed these results with the default and customized value for "**`UV_THREADPOOL_SIZE`**".

The following code is used for the preceding experiment with "**`UV_THREADPOOL_SIZE`**":

```
1. const z = require('zlib')
2. const c = require('crypto')
3. const p = require('perf_hooks')
4. const start = p.performance.now()
5.
6. process.on('exit', (c) => {
7.   const end = p.performance.now()
8.   console.log(Math.round(end - start))
9. })
10.
11. for(var i = 0; i< +process.argv[2]; i++) {
12.   c.randomBytes(1024 * 1024, (e, b) => {
13.     z.gzip(b, (r, d) => {})
14.   })
15. }
```

The preceding code creates random bytes of 1MB and compresses them. The time taken for this action is measured using performance counter APIs.

The following output shows a 40% reduction in CPU time with thread pool size tuning:

```
01. $ node pool.js 1000
02. 11534
03. $ export UV_THREADPOOL_SIZE=8
04. $ node pool.js 1000
05. 7216
```

Figure 10.10: *Performance improvement with thread pool tuning*

As we can see from the program output, the compression of 1MB random bytes was performed 1000 times, first with the default thread pool size that took 11.534 seconds, and then by running the same code with the thread pool size set to 8. This time, the code took 7.216 seconds to run.

Note: If you are adjusting UV_THREADPOOL_SIZE for your application, ensure that you accommodate this as well in the CPU selection of the hardware that we discussed in the hardware section.

Remove debug options

Remove any code that was used for debugging, and also remove any flags used for debugging. Options meant to be used in debugger's context are not designed to take performance into consideration and are inherently slow. This includes any tracing options like "—trace-gc" and "—trace".

Performance best practice: Application

Now that we examined the runtime optimizations, let's focus on the potential opportunities within the application itself. We will look at the application as an integration of code, data, and configuration and see how each of this can be optimized to improve our web application's performance.

Content optimization

The key to the application's performance improvement is to minimize the response time. Here are the key questions that can be asked from the application's point of view for each exposed endpoint and see how it can be addressed:

- What is the dynamic part of your response?
- What is the source of the dynamic part of the response?
- How do we ensure that the dynamic part is reduced?
- How do we ensure that the dynamic part is reusable?

Assuming that we are sending a page or part of a page (or even a piece of server-computed data) to the client, ensure that we fully understand the type of the data: is it statically available, or is it contextual and needs computation in the request context? If the response is static, there's no reason why the request

should reach up to the web backend. Instead, it can be served as static pages from the web cache and implemented by reverse proxies.

The following diagram shows how a typical response would be a mix of static and dynamic data:

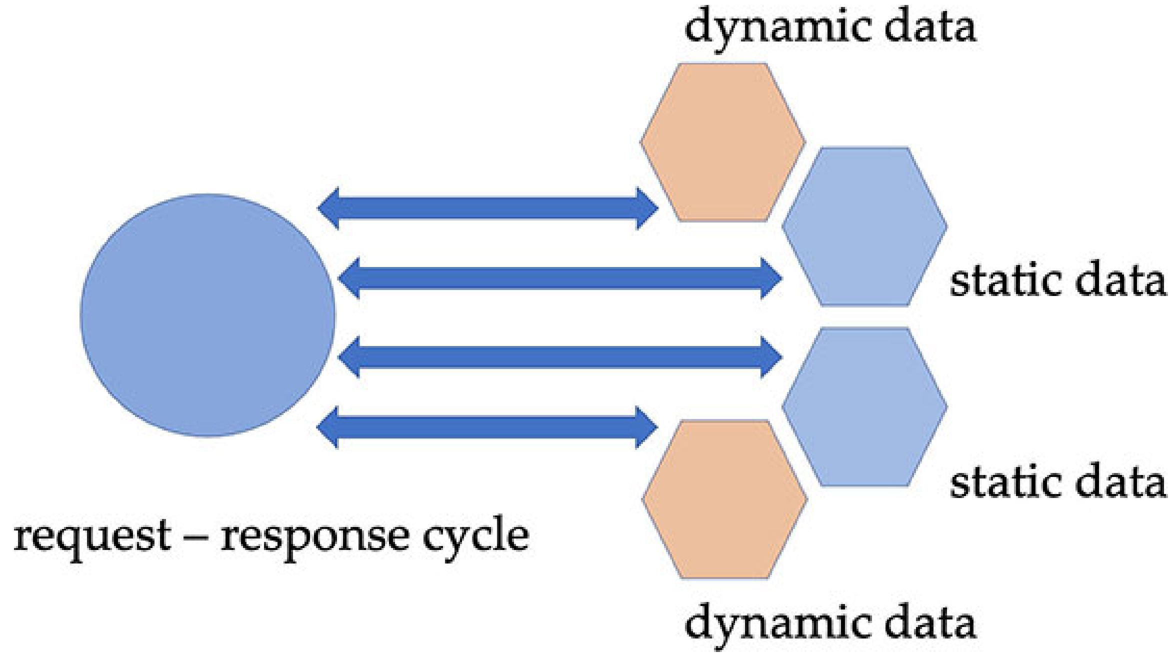

Figure 10.11: *Web response as a mixture of static and dynamic data*

Another aspect is the reusability of the dynamic part. Even if there is a dynamic part to the response, we can check if that data, in full or in parts, can be reused for the plurality of requests. That way, we are reducing the overall processing overhead on requests.

Application decoupling

When we decouple business logic, it is easy to perform the decoupling at wrong program boundaries or overdo the decoupling. Every decoupled program part (module or service) now interacts through the network instead of through direct function invocations. The net result is that the performance is affected. It is a good idea to decouple the business logic into multiple modules and services but with appropriate reasons and at reasonable code boundaries. An unnecessary decoupling costs a new network traffic in every request-response cycle.

The following diagram shows poor decoupling of the application by making a simple file reader code a separate module or service:

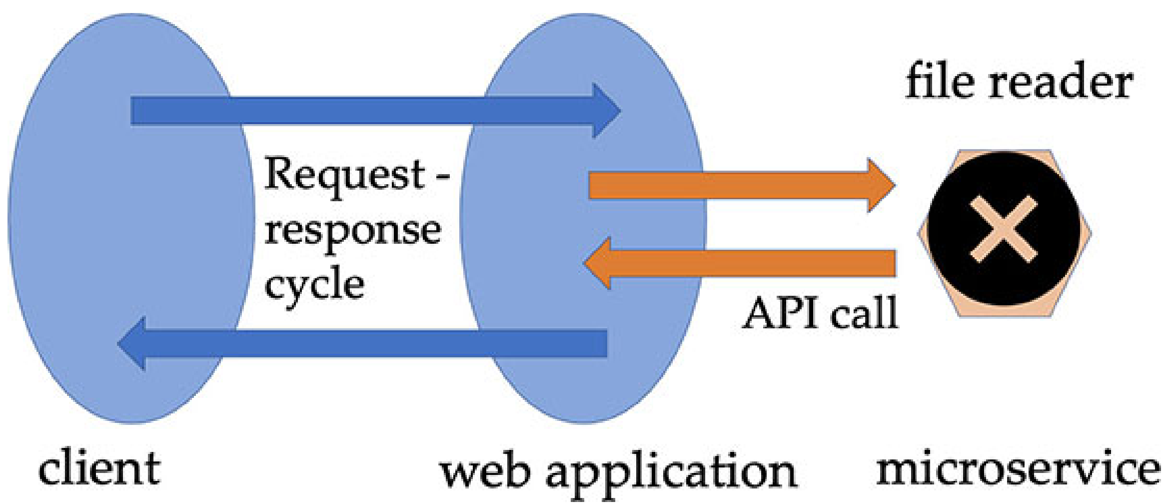

Figure 10.12: *A bad module selection for application decoupling*

And the following screenshot shows better decoupling of the application by making an inventory service a separate module:

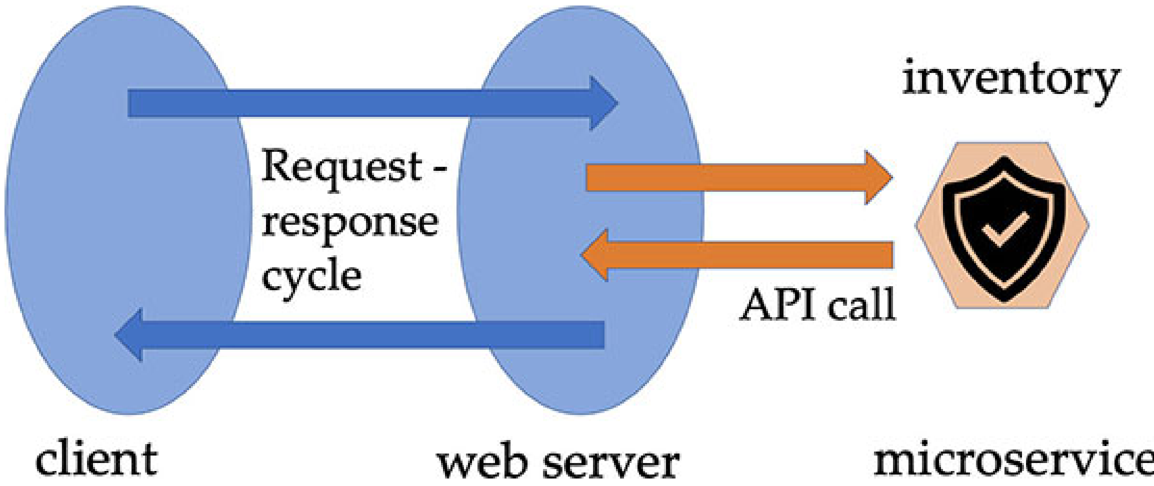

Figure 10.13: *A reasonable module selection for application decoupling*

Why can't the file reader module be decoupled from the application and made into a separate service? This is because:

- Reading a file is a relatively tiny operation that can be achieved by a direct Node.js API call
- There is no qualifier for it to be functioning as an independent service
- Large data will flow over the network if we decouple such a module.

On the other hand, the inventory service is a good candidate for decoupling due to the following reasons:

- It potentially abstracts a set of data that represents the inventory
- The service can be invoked from multiple parts of the application
- It can benefit from being a separate service; for example, it can scale up and down based on the change in requests that flow into it

HTTP caching

There are several caching possibilities throughout the request-response cycle. Some of them are part of the protocol, while some of them are configured at the reverse proxy and others are manually created and managed by the application. Use cache abundantly. A cache stores temporary data that is required frequently. Static pages, images, JavaScript sources, and CSS are natural cache candidates. It avoids file reads, copying, and parsing, which otherwise take up a good amount of CPU.

The following screenshot shows the positioning of HTTP cache in a web framework:

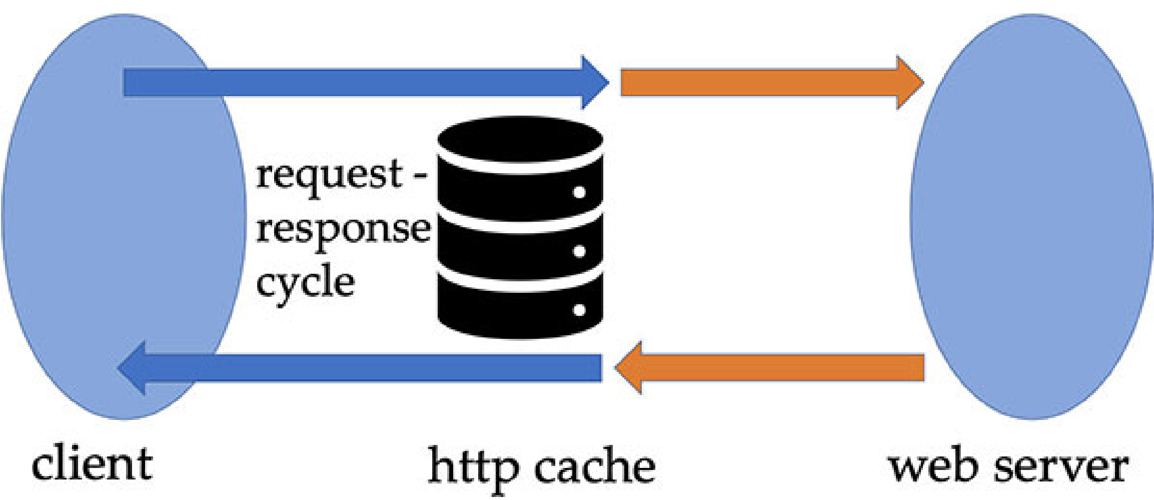

Figure 10.14: *HTTP caching model*

Ensure that the database connections are pre-created in abundance and kept in a common pool to use on demand, without being creating on every request and destroyed after every need. How do we know the number of connections to be pre-created? There are no formulae to compute that; it is a function of the number of concurrent connections and average query completion period. Arrive at a number by taking these parameters into consideration and measuring them.

The following pseudo-code shows the usage of a connection pool with a pseudo database:

```
1. dbclient.connect(url, {
2.   pool: 128,
3.   assignmentPolicy: 'ROUNDROBIN',
4.   cleanOnTimeout: true,
5.   collectStale: true,
6.   autoExpand: true,
7.   autoShrink: true
8. })
```

Optimize your database. If it is an SQL database, ensure that it is normalized and indexed. Use precise queries to reduce the pressure on both the database and the code that receives and processes the output. Identify the read-only sections of the database and qualify them as such. Additionally, cache such sections in-memory as these are read-only copies of data.

API selection

Use "*ChildProcess*" APIs only if required. Many asynchronous APIs fulfil the needs of what a program typically wants to execute through a child process (for example, file system operations, obtaining process memory, and so on). Invoking those APIs in-process is much more efficient than starting a new process altogether.

The following code lists the files in the current folder using the ChildProcess API:

```
1. const child = require('child_process')
2.
3. const ls = child.spawn('ls')
4.
5. ls.stdout.on('data', (d) => {
6.   console.log(d.toString())
7. })
8.
```

```
9. ls.stderr.on('data', (d) => {
10.   console.log(d.toString())
11. })
```

In the preceding code, a UNIX command "**ls**" is being invoked from the Node.js process via the "**spawn**" API from the "**child_process**" module that is capable of spawning a child process. The output of the child process is obtained via standard event and stream semantics. So, the directory listing is obtained and printed in the 'data' handler callback in the parent.

The following code lists files in the current folder using the "**fs**" API. The key difference is that the former uses a child process, while the latter achieves the same in-process:

```
1. const f = require('fs')
2. f.readdir('.', (e, l) => {
3.   l.forEach((e) => {
4.     console.log(e)
5.   })
6. })
```

The following output shows the time taken by the ChildProcess API for the task:

```
01. $ time node cp.js
02. foo.txt
03. bar.txt
04. baz.txt
05. zoo.txt
06. zab.txt
07.
08. real 0m0.052s
09. user 0m0.047s
10. sys 0m0.007s
```

Figure 10.15: *Time spent for child process-based execution*

Here's the output showing the time taken by the "fs" API for the same task. As is evident, the in-process model works faster:

```
01. $ time node api.js
02. foo.txt
03. bar.txt
04. baz.txt
05. zoo.txt
06. zab.txt
07.
08. real 0m0.046s
09. user 0m0.025s
10. sys 0m0.022s
```

Figure 10.16: *Time spent for API-based execution*

Consider the usage of HTTP/2 instead of HTTP, as applicable. This is an improvement over HTTP and enhances data transport through HTTP header compression. Plus, features like pro-active push from the server rather than pulling from the clients. Of course, switching the protocol warrants code changes to that effect as well.

The following code snippet shows the usage of the HTTP2 module for secure and flexible communication:

```
1. const h = require('http2')
2.
3. const s = h.createSecureServer({key: key, cert: cert})
4. s.on('stream', (r, h) => {
5.   s.respond({'content-type': 'text/html',':status': 200})
6.   s.end('http2')
7. })
8. s.listen(12000)
```

Almost all data emanating from the I/O is stream-oriented. If such data is destined to I/O targets, ensure that it is not buffered in memory but streamed

with an optional transformer or filter as required.

The following code shows working with flowing data without buffering it in intermediary memory. The natural flow rate is maintained, and the data is processed in-flight:

```
1. const s = require('stream')
2. const f = require('fs')
3.
4. const t = new s.Transform({
5.   transform(d, e, c) {
6.     c(null, d.toString().toUpperCase())
7.   }
8. })
9. const d = f.createReadStream(__filename)
10. d.pipe(t)
11. t.pipe(process.stdout)
```

In the preceding example, there are three streams: the read stream that we get when we open the file in streaming mode; **`process.stdout`**, which is a write stream with the console as the target; and a transform stream (read-write stream) that we created to read from the file reader, apply some transformation and then write into the console. As we can see, the transformation is applied on the incoming data as chunks. Using this approach, we can ensure that the incoming data is not buffered internally.

Miscellaneous

Avoid explicit invocation of garbage collection (gc). Once the JavaScript heap is sufficiently sized, leave the garbage collection decision to the runtime engine instead of explicitly invoking gc. The programming context that you selected to invoke gc and the internal context that the runtime engine maintains to manage the garbage can be mutually conflicting in their functioning, so mixing both can make your application highly inefficient, leading to suboptimal performance.

The following code shows a plain loop computation in action:

```
1. let l = 0
```

```
2. for(var i = 0; i < +process.argv[2]; i++) {
3.   l += i
4. }
```

And the following output shows the time taken for executing that loop:

```
01. $ time node gc.js 5000
02.
03. real 0m0.069s
04. user 0m0.048s
05. sys 0m0.013s
```

Figure 10.17: *Time for executing a trivial 'for' loop*

The following code shows the same loop, but invoking garbage collector explicitly:

```
1. let l = 0
2. for(var i = 0; i < +process.argv[2]; i++) {
3.   l += i
4.   gc()
5. }
```

And the following output shows the time taken for executing that loop with garbage collection within the loop. Clearly, we are wasting a lot of CPU cycles by invoking garbage collection where potentially no garbage is present. Depending on the loop counter variable, the wasted CPU cycles can be minimal, substantial, or exorbitant:

```
01. $ time node --expose-gc gc.js 5000
02.
03. real 0m8.477s
04. user 0m16.757s
05. sys 0m0.961s
```

Figure 10.18: *Time for executing a for loop with gc calls in it*

If a code flow is identified as always throwing exception and being caught in the bottom of the stack, the whole sequence is irrelevant and adds no value to the application's business logic. The whole flow can be bypassed.

The following code shows a bad way of sending data to a remote endpoint, by trying on an arbitrary connection and rectifying the issue upon an exception:

```
1. try {
2.   db.update(data)
3. } catch(e) {
4.   if (e.message === 'StaleConnectionError') {
5.     db.reconnect()
6.     db.update(data)
7.   }
8. }
```

And the following code shows a good way of achieving the same with better performance, as we are avoiding an exception unwinding and stack walk, which are expensive from the CPU standpoint, in addition to several CPU cycles being wasted in the failed attempt:

```
1. if (db.connection.status === db.CONN.STALE)
2.   db.reconnect()
3. db.update(data)
```

Reading from a file is a costly operation, so avoid it in the request-response path as much as possible. If the file is static, read it once and store it in memory. Implement a file watcher, watch the file for modifications, and refresh the cached content if the file content changes dynamically at a lower frequency (than the request frequency).

The following code shows how to manage static and semi-static file content without performing a disk I/O in each request:

```
1. const s = require('stream')
2. const f = require('fs')
3. const h = require('http')
4.
5. const t = new s.Transform({
6.   transform(d, e, c) {
7.     c(null, d.toString().toUpperCase())
```

```
8.   }
9. })
10. const file = './tmp.data'
11. let d = f.readFileSync(file)
12. f.watch(file, (e, g) => {
13.   d = f.readFileSync(file)
14. })
15. h.createServer((q, r) => {
16.   t.pipe(r)
17.   t.write(d)
18.   r.end()
19. }).listen(12000)
```

In the preceding example, we read from file content once and cache it in a variable. However, we install a file watcher as there is no guarantee that the file is static. The watcher watches for file content changes and rereads from the file and replenishes the cache in response to any change, ensuring that the file is not read on each request and that the data is updated whenever there is a file content change.

If the size of the response data is more than 64KB, (or whatever the socket buffer and operating system buffer can hold) consider compressing the response with “gzip”, which is a fast compression algorithm, and the protocol is well understood by networking endpoints. Doing so reduces the network traffic and saves additional round trips. On the other hand, avoid compression for small amounts of data. Either way, do not mix compression strategy in a single endpoint; instead, fix on one based on the common nature and size of the data flow from that endpoint.

The following screenshot shows a typical request header with transfer encoding:

```
01. GET / HTTP/1.1
02. Host: localhost:12000
03. Transfer-Encoding: gzip
04. Connection: close
```

Figure 10.19: *HTTP request header that uses transfer encoding*

Where to store session data? The answer to this varies. If your application uses replicated instances with a reverse proxy to balance the load and perform the routing and your priority is to provide high availability, the session data can be kept in a central store that is accessible from all the replicas. That way, even if an instance crashes, the other instances can process the request with the help of the central session data, making the crash invisible to the client. The drawback of this approach is that it impacts the performance for all transactions. This is because for every request, there are at least two additional I/O involved to fetch the session data upon entry and store the session data back upon exit. On the other hand, an in-memory session will be a good choice if you are focused on improving performance and not worried about call drops in case the server instance crashes.

What to store in a session? Ideally, all contextual data pertinent to the current user session that will not have side effects on the rest of the system should be stored. On the other hand, ensure that a huge amount of session data does not explode your memory.

The following snapshot shows session data:

```
01. session: {
02.   id: "dse23nbj322fftyuh290oi13d",
03.   username: "ijk671987",
04.   logintime: 1652323992122,
05.   lastPageAccess: "inventory",
06.   cart: [
07.     "apple-2324", "grapes-181"
08.   ]
09. }
```

Figure 10.20: *An example session data*

> **Note:** The session data is a per user request object. This means the effective (memory) pressure exerted on the runtime engine is equal to the product of individual session data and the number of concurrent users operated with the session data.

Call "**`require()`**" once rather than in a re-enterable block. Module loading is a very complex and CPU-consuming activity, and if you require a module inside block repeatedly, it will run a lot of code before it realizes that the module is already loaded, wasting CPU cycles in the process.

The following code shows "**`require`**" being invoked in re-enterable code blocks:

```
1. function addID(d) {
2.   let b = require('v8').deserialize(d)
3.   b.id = require('crypto').randomUUID()
4.   return require('v8').serialize(d))
5. }
```

And the following code shows a better way of requiring modules:

```
1. const c = require('crypto')
2. const v = require('v8')
3.
4. function addID(d) {
5.   let b = v.deserialize(d)
6.   b.id = c.randomUUID()
7.   return v.serialize(d))
8. }
```

Ensure that highly computational activities do not come in the way of the application thread. If inevitable, off-load them to a worker thread or a child process, whichever is convenient. As a thumb rule, offload anything that takes more than 10ms. The thumb rule here is to off-load any scratch work asynchronously to the worker threads.

Conclusion

In this chapter, we examined web application performance, which is one of the key attributes that make a production-grade website. We examined various components involved in the hardware and software stack of the application and identified best practices for tuning each one to improve the overall performance. We also discussed some deployment options that build up the execution environment of the application, as several performance best practices are tightly coupled with the way the application is deployed.

In the next chapter, we will look at problem determination best practices for the most common production anomalies that can occur in a highly concurrent web application. Knowing how to detect, diagnose, and fix those anomalies will complete our learning of a web application.

Questions

1. What is old space and new space in the JavaScript heap? What data do they store? Where do the application's objects reside?
2. Some operations, such as pure I/O operations, are made asynchronous with direct support from the operating system, without needing to adjust the thread pool size. On the other hand, some operations, such as compression, are simulated with additional helper threads. Can you identify the fundamental reason for this differential treatment?
3. Discuss the advantages and disadvantages of buffered operations on network data in synchronous and asynchronous programming models.

CHAPTER 11
Debugging Program Anomalies

In the last chapter, we examined web application performance, in continuation with the previous chapters, wherein we discussed other attributes that make a production-grade website. We took up various programming scenarios and made observations and inferences around the best practices to be followed in dealing with web application workloads. In this chapter, we will look at problem determination (troubleshooting) best practices for the most common production anomalies that can potentially occur in a highly concurrent web application. Knowing how to detect, diagnose, and fix these anomalies will complete our learning of a web application and its long-term maintenance.

Structure

We will cover the following topics in this chapter:

- Debugging crash
- Debugging low CPU (hang)
- Debugging high CPU (spin)
- Debugging memory leak/exhaustion
- Debugging performance degradation

Objective

After studying this chapter, you will be able to understand tools, frameworks, methodologies, and best practices for performing basic problem determination of our web application. You will be able to observe the application's external symptoms and make assertions about the type of anomaly, devise a debugging strategy accordingly, pick the right set of tools and methodologies, and troubleshoot the issue yourself. If the issue is rooted deeper inside the application stack, such as Node.js core or the

operating system, you will be able to isolate it to the extent of providing a first-hand report and the steps to reproduce the problem so as to pass it to the concerned team and get it rectified in a timely manner. Some of these are generic learnings that you can apply on other platforms/applications as well.

Debugging crash

While there can be multiple definitions for a program crash depending on the nature of the application and the referring person, there are at least two broad types of crashes:

- Processor instigated program abends
- Software instigated program abends

Here, processor means the CPU, and abend means an abnormal ending of the process. The term "*program crash*" was traditionally used to refer to the first category, but over time, any abnormal program termination, regardless of its nature and origin, was termed as a crash.

An example of the first category (CPU or processor instigated crash) is a typical program abend with the program receiving a signal, such as "SIGSEGV". An example of the second category (program instigated crash) is an application termination with an exception. Exceptions can be generally thrown from the application itself or from the lower-level stack (such as the middleware, Node.js runtime API, or the C or C++ runtime libraries). When the said exception does not have a handler function to absorb, handle, and contain the exception, it results in an unhandled exception and crashes.

The following snapshot shows a Node.js program crash with segmentation fault:

```
01. $ node app.js
02. [1]    176239 segmentation fault  /usr/local/bin/node
```

Figure 11.1: *An application crash with SIGSEGV*

An example of the second category (program instigated crash) is a program abend with the Node.js runtime asserting when it encounters an unexpected scenario.

The following snapshot shows a Node.js program crash with assertion failure:

```
01. $ node app.js
02. /usr/local/bin/node[13458]:
03. ../src/node.cc:649:
04. void node::ResetStdio():
05. Assertion `(err) != (-1)` failed.
```

Figure 11.2: *Application crash with low-level assertion failure*

In the preceding example, the Node.js runtime is throwing an assertion because it encountered an unexpected program state. While resetting the standard IO handles, it applied file control operations ("fcntl" system call) on the file descriptor of each handle and made an assertion that the return value of the call should NOT be -1; the system call should not fail for this handle. In our erroneous case, the assertion was hit. As it is evident from the assertion, the program cannot make meaningful progress with the system call fails on such as file descriptor – the file descriptor that the program believes is in good shape and want to use it for further operations.

Yet another example of the second category (program instigated crash) is a program abend with the application throwing an exception when a NULL value is treated as a function and an invocation is attempted on it after a foo function is invoked.

The following code shows a sample application function that throws an exception:

```
1. // name: foo
2. // function: write data into the object's
3. // internal buffer
4. // invoke a special handler for long writes
5. function foo(data, encoding, cb) {
6. if (data.length > 64 * 1024) {
7. return longWrite(data, encoding, cb)
8. }
```

```
9. setTimeout(() => {
10. this.buffer.push(data.toString())
11. this.written += data.length
12. cb()
13. }, Math.floor(Math.random() * 128))
14. }
```

The following snapshot shows this program's output when crashed with exception:

```
01. $ node app.js
02.
03. Typeerror: cb is not a function
04. at foo (/usr/lib/node_modules/foo/foo.js:12:13)
05. at bar (/usr/lib/node_modules/bar/bar.js:24:45)
06. at onEvent(/usr/lib/node_modules/handler/handler.js:78:342)
07. at emit(events.js:198:13)
```

***Figure 11.3:** Application crash with exception*

Preparation

It is not easy to reproduce the crash a second time, on will. For that matter, it is always recommended to prepare your host system (or systems) to produce all the intended diagnostic data upon the first failure itself. The common preparatory steps for a Node.js web application toward crash diagnostics are as follows:

1. Change the relevant operating system settings so that a core dump (the purpose of the core dump will become evident when we visit the problem determination section) can be produced upon abnormal program conditions.

   The following command enables core dump generation and defines a location in Linux:

   ```
   1. $ echo "/home/nodeuser/dump/core.%e.%p" > /proc/sys/kernel/core_pattern
   ```

2. Change the relevant user limits on the system so that a core dump can be produced and written to the disk.

The following command enables disk space for core dump generation:

1. `$ ulimit -c unlimited`

3. Configure the application command line arguments so that it is forced to abort, leading to the generation of a core dump upon the event of an uncaught exception.

   The following command enables the process to generate a core dump on uncaught exceptions:

   1. `$ node --abort-on-uncaught-exception app.js`

4. Configure the application command line argument so that it produces a diagnostic report upon the event of an uncaught exception or an abnormal program termination.

   The following command enables the process to generate a report on uncaught exceptions:

   1. `$ node --report-uncaught-exception app.js`

Symptom

If you are using a single instance of the backend application, the frontend will report the status code 404 or an equivalent error message to indicate that the server is unavailable.

The following snapshot shows the client browser when its backend has crashed:

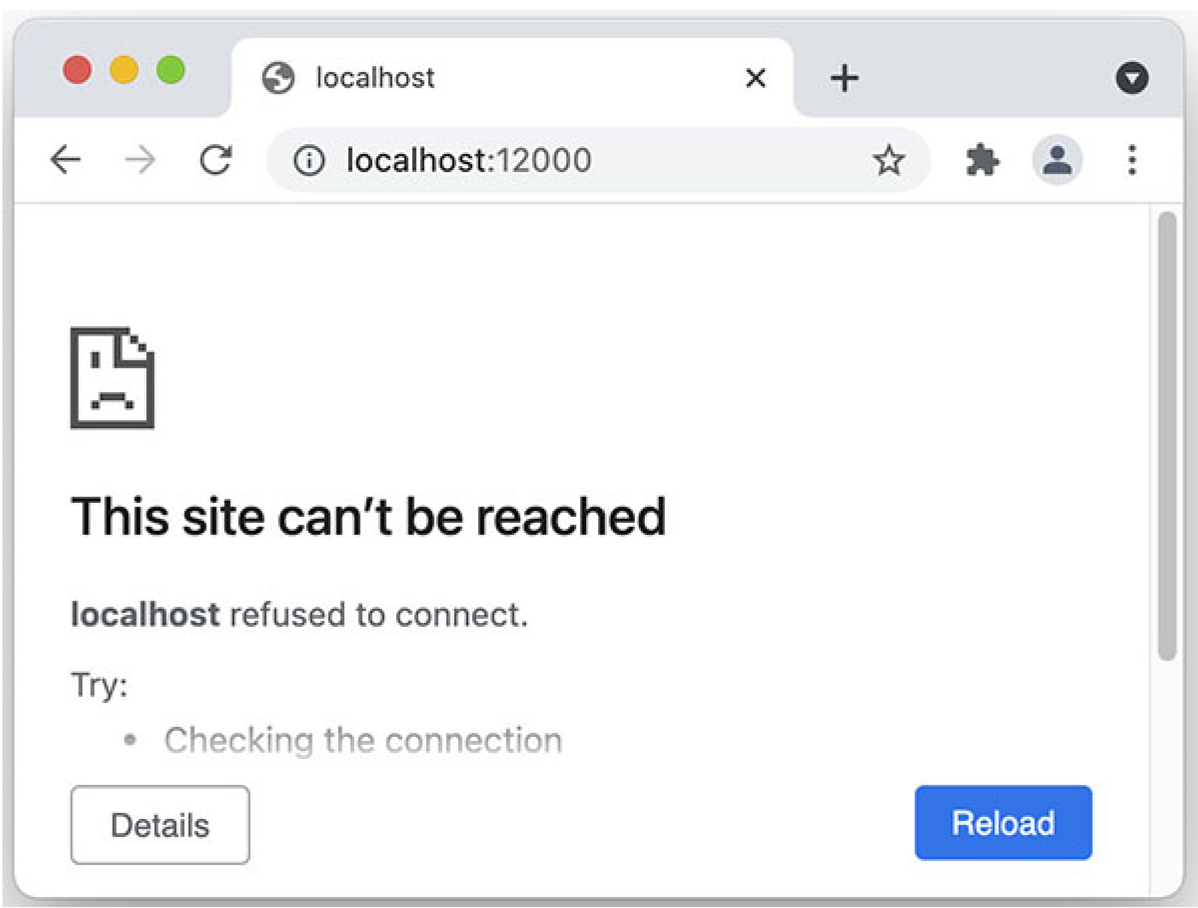

Figure 11.4: *External manifestation of a server crash*

If you are using multiple replicas of your application and one of the instances crashes, it's effects may not be visible from the frontend as the request may be handled by other healthy replicas, unless all the replicas go down together.

On the other hand, the crash may not be easily visible but a few requests may get dropped off if you are using an automatic recycler program script which immediately brings a crashed program back to life.

However, in the server systems, the log file will show messages pertinent to the exceptions, program assertions, or abend signals that affected the program and led to the crash in all cases.

Useful data

If the application crashed due to processor-instigated crash (CPU signal), the most important data is the core dump (also known as system dump or

abend dump).

If the application crashed due to assertion failures, the most important data is the core dump and the assertion failure message itself.

If the application crashed due to environmental error conditions, such as memory corruption or memory exhaustion scenarios, the `stdout` log and the memory dump (heap dump, core dump) are useful.

If the application crashed due to exceptions, the `stdout` log is the most useful data.

In all cases, the diagnostic report is a good **First-Failure-Data-Capture** (**FFDC**) document as it provides a starting point for the debugging path. In many cases, the diagnostic report alone is sufficient to perform the root cause analysis and fix the issue.

Problem determination

The methodology of problem determination is approximately the same in all types of crashes, but the tools and specific steps may differ. The established and recommended way of debugging a program crash is as follows:

- Identify the immediate reason for the crash - at the instruction, expression, or statement level (depending on the nature of the crash)
- Identify the operation (code) and the operands (code) involved in the crash
- The procedure is over if the reason directly leads to the identification of the issue
- Else, walk backward in the block of code and function following the call stack and interfering threads by tracing the trajectory of the issue in the code and/or data
- Repeat he previous step until the root cause is identified

Generally, the root cause of a crash is one or more of the following:

- Bad code
- Bad data
- Bad input

- Bad timing

A core dump contains the following data points to help diagnose the issue. A debugger tool that understands the core dump format can extract this useful data:

- Failing instruction
- Register values at the failing context
- Call stack at the failing context
- Complete program memory dump
- State of all threads in the process
- Loaded library information

The following code is used to explain the problem determination with the help of a crash dump:

```
1. sock.on('data', (d) => {
2. console.log(d.toString())
3. })
```

A set of data points is collected and explained in the crash dump pertinent to the previous program. Evidently, the code is fine, but some issues in the lower-level stack (the API, runtime engine, and operating system) caused the crash.

The following snapshot shows the *Instruction Address Register* content in a core dump:

```
01. (gdb) info registers $rip
02. rip            0x7f15715807ef      0x7f15715807ef <epoll_wait+79>
```

Figure 11.5: *Instruction Address Register in gdb*

The following snapshot shows all the CPU registers of the failing thread in a core dump:

```
01.  (gdb) info registers
02.  rax            0xfffffffffffffffc   -4
03.  rbx            0x9                  9
04.  rcx            0x7f15715807ef       139730072635375
05.  rdx            0x400                1024
06.  rsi            0x7f157147dd80       139730071575936
07.  rdi            0x9                  9
08.  rbp            0x7f157147dd80       0x7f157147dd80
09.  rsp            0x7f157147dc60       0x7f157147dc60
10.  r8             0x0                  0
11.  r9             0x0                  0
12.  r10            0xffffffff           4294967295
13.  r11            0x293                659
14.  r12            0x400                1024
15.  r13            0xffffffff           4294967295
16.  r14            0x5016100            83976448
17.  r15            0x0                  0
18.  rip            0x7f15715807ef       0x7f15715807ef <epoll_wait+79>
19.  eflags         0x293                [ CF AF SF IF ]
20.  cs             0x33                 51
21.  ss             0x2b                 43
22.  ds             0x0                  0
23.  es             0x0                  0
24.  fs             0x0                  0
25.  gs             0x0                  0
```

Figure 11.6: *CPU register values of the current thread in gdb*

The following snapshot shows the current thread's call stack in a core dump:

```
01.  (gdb) where
02.  #0  0x7f15715807ef in epoll_wait (epfd=epfd@entry=9,
03.      events=events@entry=0x7f157147dd80, maxevents=maxevents@entry=1024,
04.      timeout=timeout@entry=-1) at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
05.  #1  0x00000000015d1b04 in uv__io_poll (loop=loop@entry=0x5016168,
06.      timeout=<optimized out>) at ../deps/uv/src/unix/linux-core.c:324
07.  #2  0x00000000015bf2f8 in uv_run (loop=0x5016168, mode=UV_RUN_DEFAULT)
08.      at ../deps/uv/src/unix/core.c:385
09.  #3  0x0000000000b9391b in
10.      node::WorkerThreadsTaskRunner::DelayedTaskScheduler::Start()::{lambda(void*)#1}::_FUN(void*) ()
11.      at ../deps/uv/src/uv-common.c:887
12.  #4  0x00007f157164ffa3 in start_thread (arg=<optimized out>)
13.      at pthread_create.c:486
14.  #5  0x00007f15715804cf in clone ()
15.      at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95
```

Figure 11.7: *Call stack of the current thread in gdb*

The following snapshot shows the memory mappings of the process in a core dump:

```
01.  (gdb) info proc mappings
02.  Mapped address spaces:
03.
04.            Start Addr           End Addr       Size     Offset objfile
05.              0x400000          0x4653000  0x4253000        0x0 /usr/local/bin/node
06.             0x4852000          0x4856000     0x4000  0x4252000 /usr/local/bin/node
07.             0x4856000          0x486a000    0x14000  0x4256000 /usr/local/bin/node
08.             0x486a000          0x488f000    0x25000        0x0
09.             0x4fd3000          0x51e8000   0x215000        0x0 [heap]
10.        0x7f1571487000     0x7f15714a9000    0x22000        0x0 /lib/x86_64-linux-gnu/libc-2.28.so
11.        0x7f1571648000     0x7f157164e000     0x6000        0x0 /lib/x86_64-linux-gnu/libpthread-2.28.so
12.        0x7f1571669000     0x7f157166c000     0x3000        0x0 /lib/x86_64-linux-gnu/libgcc_s.so.1
13.        0x7f1571683000     0x7f1571690000     0xd000        0x0 /lib/x86_64-linux-gnu/libm-2.28.so
14.        0x7f1571806000     0x7f157188f000    0x89000        0x0 /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
15.        0x7f157198a000     0x7f157198b000     0x1000        0x0 /lib/x86_64-linux-gnu/libdl-2.28.so
16.        0x7f157199a000     0x7f157199b000     0x1000        0x0 /lib/x86_64-linux-gnu/ld-2.28.so
17.        0x7ffffe270000     0x7ffffe291000    0x21000        0x0 [stack]
18.        0x7ffffe378000     0x7ffffe37c000     0x4000        0x0 [vvar]
19.        0x7ffffe37c000     0x7ffffe37e000     0x2000        0x0 [vdso]
20.    0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

Figure 11.8: *Memory regions of the process in gdb*

The following snapshot shows the state of all the threads in a core dump:

```
01.  (gdb) info threads
02.    Id   Target Id                             Frame
03.    1    Thread 0x7f15714827c0 (LWP 30) "node" 0x7f15714bea97 in kill ()
04.      at ../sysdeps/unix/syscall-template.S:78
05.  * 2    Thread 0x7f1571481700 (LWP 31) "node" 0x7f15715807ef in epoll_wait (
06.      epfd=epfd@entry=9, events=events@entry=0x7f157147dd80,
07.      maxevents=maxevents@entry=1024, timeout=timeout@entry=-1)
08.      at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
09.    3    Thread 0x7f1570c80700 (LWP 32) "node" futex_wait_cancelable (private=0,
10.      expected=0, futex_word=0x5015d40)
11.      at ../sysdeps/unix/sysv/linux/futex-internal.h:88
12.    4    Thread 0x7f156bfff700 (LWP 33) "node" futex_wait_cancelable (private=0,
13.      expected=0, futex_word=0x5015d40)
14.      at ../sysdeps/unix/sysv/linux/futex-internal.h:88
15.    5    Thread 0x7f156b7fe700 (LWP 34) "node" futex_wait_cancelable (private=0,
16.      expected=0, futex_word=0x5015d40)
17.      at ../sysdeps/unix/sysv/linux/futex-internal.h:88
18.    6    Thread 0x7f156affd700 (LWP 35) "node" futex_wait_cancelable (private=0,
19.      expected=0, futex_word=0x5015d40)
20.      at ../sysdeps/unix/sysv/linux/futex-internal.h:88
21.    7    Thread 0x7f1571999700 (LWP 36) "node" futex_abstimed_wait_cancelable (
22.      private=0, abstime=0x0, expected=0,
23.      futex_word=0x486bdc0
24.      <node::inspector::(anonymous namespace)::start_io_thread_semaphore>)
25.      at ../sysdeps/unix/sysv/linux/futex-internal.h:205
26.    8    Thread 0x7f15627fc700 (LWP 37) "node" futex_wait_cancelable (private=0,
27.      expected=0, futex_word=0x487724c <cond+44>)
28.      at ../sysdeps/unix/sysv/linux/futex-internal.h:88
29.    9    Thread 0x7f1561ffb700 (LWP 38) "node" futex_wait_cancelable (private=0,
30.      expected=0, futex_word=0x4877248 <cond+40>)
31.      at ../sysdeps/unix/sysv/linux/futex-internal.h:88
32.    10   Thread 0x7f15617fa700 (LWP 39) "node" futex_wait_cancelable (private=0,
33.      expected=0, futex_word=0x4877248 <cond+40>)
34.      at ../sysdeps/unix/sysv/linux/futex-internal.h:88
35.    11   Thread 0x7f1560ff9700 (LWP 40) "node" futex_wait_cancelable (private=0,
36.      expected=0, futex_word=0x487724c <cond+44>)
37.      at ../sysdeps/unix/sysv/linux/futex-internal.h:88
```

Figure 11.9: Information of the process threads in gdb

The following snapshot shows the loaded shared libraries in a core dump:

```
01.  (gdb) info sharedlibrary
02.  From                To                  Syms Read   Shared Object Library
03.  0x7f157199b090  0x7f15719b8b20  Yes         /lib64/ld-linux-x86-64.so.2
04.  0x7f157198b130  0x7f157198be75  Yes         /lib/x86_64-linux-gnu/libdl.so.2
05.  0x7f15718924b0  0x7f157193a04e  Yes (*)     /usr/lib/x86_64-linux-gnu/libstdc++.so.6
06.  0x7f1571690270  0x7f157172e4f2  Yes         /lib/x86_64-linux-gnu/libm.so.6
07.  0x7f157166c2e0  0x7f157167cc2d  Yes (*)     /lib/x86_64-linux-gnu/libgcc_s.so.1
08.  0x7f157164e5b0  0x7f157165c641  Yes         /lib/x86_64-linux-gnu/libpthread.so.0
09.  0x7f15714a9320  0x7f15715ef39b  Yes         /lib/x86_64-linux-gnu/libc.so.6
```

***Figure 11.10:** Information on the loaded libraries in gdb*

A debugger tool that understands both the core dump format and the internal data structures of the Node.js runtime can extract this additional data:

- JavaScript objects
- JavaScript heap
- Node.js internal data structures
- Composite call stack (C, C++ and JavaScript)

A diagnostic report contains the following data points to help diagnose the issue:

- Abend trigger reason
- Abend time stamp
- Process ID
- Process command line
- Version of internal components
- CPU, OS, and architecture information
- JavaScript stack (JS)
- Native stack (C, C++)
- JavaScript heap sections
- Resource consumption details
- Handles in the event loop
- Pending timers
- Environment variables

- User resource limits
- Loaded shared libraries

The following snapshot shows the header section of a diagnostic report:

```
01.   "header": {
02.     "reportVersion": 2,
03.     "event": "Cannot read property 'cb' of undefined",
04.     "trigger": "Exception",
05.     "filename": "report.20210821.085234.49.0.001.json",
06.     "dumpEventTime": "2021-08-21T08:52:34Z",
07.     "dumpEventTimeStamp": "1629535954683",
08.     "processId": 49,
09.     "threadId": 0,
10.     "cwd": "/",
11.     "commandLine": [
12.       "node",
13.       "--report-uncaught-exception",
14.       "app.js"
15.     ]
16. }
```

Figure 11.11: *Diagnostic report – header section*

The following snapshot shows execution environment-related data in a diagnostic report:

```
01.     "nodejsVersion": "v16.5.0",
02.     "glibcVersionRuntime": "2.28",
03.     "glibcVersionCompiler": "2.17",
04.     "wordSize": 64,
05.     "arch": "x64",
06.     "platform": "linux",
07.     "componentVersions": {
08.       "node": "16.5.0",
09.       "v8": "9.1.269.38-node.20",
10.       "uv": "1.41.0",
11.       "zlib": "1.2.11",
12.       "brotli": "1.0.9",
13.       "ares": "1.17.1",
14.       "modules": "93",
15.       "nghttp2": "1.42.0",
16.       "napi": "8",
17.       "llhttp": "6.0.2",
18.       "openssl": "1.1.1k+quic",
19.       "cldr": "39.0",
20.       "icu": "69.1",
21.       "tz": "2021a",
22.       "unicode": "13.0",
23.       "ngtcp2": "0.1.0-DEV",
24.       "nghttp3": "0.1.0-DEV"
25.     }
```

Figure 11.12: *Diagnostic report – version information*

The following snapshot shows the JavaScript stack in a diagnostic report:

```
01.   "javascriptStack": {
02.       "TypeError: Cannot read property 'cb' of undefined",
03.       "at [eval]:1:13",
04.       "at Script.runInThisContext (node:vm:129:12)",
05.       "at Object.runInThisContext (node:vm:305:38)",
06.       "at node:internal/process/execution:81:19",
07.       "at [eval]-wrapper:6:22",
08.       "at evalScript (node:internal/process/execution:80:60)"
09.     ],
10.     "errorProperties": {
11.     }
12.   }
```

Figure 11.13: *Diagnostic report – JavaScript stack*

The following snapshot shows the JavaScript heap section in a diagnostic report:

```
01.   "javascriptHeap": {
02.     "totalMemory": 4251648,
03.     "totalCommittedMemory": 3355800,
04.     "usedMemory": 3359752,
05.     "availableMemory": 1065274600,
06.     "memoryLimit": 1067712512,
07.     "heapSpaces": {
08.       "old_space": {
09.         "memorySize": 2650112,
10.         "committedMemory": 2613856,
11.         "capacity": 2598552,
12.         "used": 2598552,
13.         "available": 0
14.       },
15.       "new_space": {
16.         "memorySize": 1048576,
17.         "committedMemory": 265912,
18.         "capacity": 1031072,
19.         "used": 248408,
20.         "available": 782664
21.       }
22.     }
23.   }
```

Figure 11.14: *Diagnostic report – JavaScript heap*

The following snapshot shows the resource usage section in a diagnostic report:

```
01.   "resourceUsage": {
02.     "userCpuSeconds": 0.028517,
03.     "kernelCpuSeconds": 0.01324,
04.     "cpuConsumptionPercent": 4.1757,
05.     "maxRss": 32428032,
06.     "pageFaults": {
07.       "IORequired": 1,
08.       "IONotRequired": 2252
09.     },
10.     "fsActivity": {
11.       "reads": 72,
12.       "writes": 0
13.     }
14.   }
```

Figure 11.15: *Diagnostic report – resource usage*

The following snapshot shows the event loop and its handles in a diagnostic report:

```
01.  "libuv": [
02.    {
03.        "type": "async",
04.        "is_active": true,
05.        "is_referenced": false,
06.        "address": "0x0000000005dc1100"
07.    },
08.    {
09.        "type": "idle",
10.        "is_active": false,
11.        "is_referenced": true,
12.        "address": "0x0000000005e80cf0"
13.    }
14.  ]
```

Figure 11.16: *Diagnostic report – handles and their states in the event loop*

The following snapshot shows the environment variables in a diagnostic report:

```
01.  "environmentVariables": {
02.    "HOSTNAME": "cfw656sa",
03.    "YARN_VERSION": "1.22.5",
04.    "PWD": "/",
05.    "HOME": "/root",
06.    "TERM": "xterm",
07.    "SHLVL": "1",
08.    "PATH": "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
09.    "NODE_VERSION": "16.5.0",
10.    "_": "/usr/local/bin/node"
11.  }
```

Figure 11.17: *Diagnostic report – environment variables*

The following snapshot shows the user limit settings in a diagnostic report:

```
01.   "userLimits": {
02.     "core_file_size_blocks": {
03.       "soft": "unlimited",
04.       "hard": "unlimited"
05.     },
06.     "file_size_blocks": {
07.       "soft": "unlimited",
08.       "hard": "unlimited"
09.     },
10.     "max_memory_size_kbytes": {
11.       "soft": "unlimited",
12.       "hard": "unlimited"
13.     },
14.     "open_files": {
15.       "soft": 1048576,
16.       "hard": 1048576
17.     },
18.     "stack_size_bytes": {
19.       "soft": 8388608,
20.       "hard": "unlimited"
21.     },
22.   }
```

Figure 11.18: *diagnostic report – configured user limits*

The following snapshot shows the loaded shared libraries in a diagnostic report:

```
01.    "sharedObjects": [
02.      "linux-vdso.so.1",
03.      "/lib/x86_64-linux-gnu/libdl.so.2",
04.      "/usr/lib/x86_64-linux-gnu/libstdc++.so.6",
05.      "/lib/x86_64-linux-gnu/libm.so.6",
06.      "/lib/x86_64-linux-gnu/libgcc_s.so.1",
07.      "/lib/x86_64-linux-gnu/libpthread.so.0",
08.      "/lib/x86_64-linux-gnu/libc.so.6",
09.      "/lib64/ld-linux-x86-64.so.2"
10.    ]
```

Figure 11.19: *Diagnostic report – loaded shared libraries*

Evidently, a diagnostic report provides an exhaustive set of data and in most debugging scenarios, the information from the report will be a good starting point. In many cases, this information alone is sufficient to fix the issue as well.

Multi-thread interference does not usually happen in a Node.js application as the application runs on a single thread, and worker threads run in separate "v8" execution contexts. However, it is possible as the runtime itself is multi-threaded, but it occurs rarely.

On the other hand, a Node.js application suffers from a new type of issue: the call stack will not carry the complete sequence of methods that led to the crash in most scenarios of crash due to exceptions. This is due to the event-driven asynchronous programming style of Node.js. Almost all the activities in a Node.js application are event-driven, which means the bottom of the stack will be an event that triggered the current actions, and further history of the sequences behind the said event will not be available in the current call stack, limiting debugging to an extent.

The following snapshot illustrates a sample exception when an asynchronous API fails:

```
01. Error: connect ECONNREFUSED 127.0.0.1:12000
02.     at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1142:16)
03. Emitted 'error' event on ClientRequest instance at:
04.     at Socket.socketErrorListener (_http_client.js:461:9)
05.     at Socket.emit (events.js:315:20)
06.     at emitErrorNT (internal/streams/destroy.js:96:8)
07.     at emitErrorCloseNT (internal/streams/destroy.js:68:3)
08.     at processTicksAndRejections (internal/process/task_queues.js:84:21) {
09.   errno: -61,
10.   code: 'ECONNREFUSED',
11.   syscall: 'connect',
12.   address: '127.0.0.1',
13.   port: 12000
14. }
```

Figure 11.20: *Less informative call stack for asynchronous methods*

This happened when an **`http`** client tried to connect to a local server listening at port 12000. Apparently, the connection to the server is refused, and the reason is represented in the exception pertinent to the crash.

The **`async_hooks`** module that helps track the asynchronous context of the code sequence can help remedy this. The call stack generated with the help of **`async_hooks`** will form the complete asynchronous sequence, from which tracing the history backward will be easy. The resulting sequence will look the same for applications developed in any other language runtime.

The following snapshot demonstrates a sample exception when an asynchronous API fails with **`async_hooks`** used to capture and link all related asynchronous calls:

```
01. at TCPConnectWrap.afterConnect (net.js:1151:10)
02. at lookupAndConnect (net.js:1039:3)
03. at Socket.connect (net.js:975:5)
04. at Agent.connect (net.js:184:17)
05. at Agent.createSocket (_http_agent.js:287:26)
06. at Agent.addRequest (_http_agent.js:250:10)',
07. at new Resolver (internal/dns/utils.js:29:20)
08. at lookupAndConnect (net.js:1022:32)',
09. at ClientRequest.onSocket (_http_client.js:782:11)
10. at Agent.createSocket (_http_agent.js:287:26)
11. at Agent.addRequest (_http_agent.js:250:10)
12. at new ClientRequest (_http_client.js:304:16)
13. at request (http.js:46:10)
```

Figure 11.21: *Complete call stack with the help of async_hooks*

The JavaScript heap needs to be inspected with the help of a heap dump if the application crashed due to JavaScript heap memory exhaustion. Further, the dominant objects in the heap need to be mapped to their allocation context and retention context in the application, and assertions need to be made on their validity. Subsequently, a remedial action needs to be taken based on that assertion to fix the memory exhaustion issue.

The native heap needs to be inspected with the help of a core dump if the application crashed due to native heap memory exhaustion. Further, the allocation sites of memory need to be traced and matched against their de-allocation patterns, and an assertion needs to be made as to why the allocations are not getting freed or why there are so many allocations in relation to the freed ones. Subsequently, remedial actions need to be taken based on that assertion to fix the native memory issue.

Debugging low CPU

Low CPU is a situation wherein the program is slow in responding to external input. In a web application's context, the external input is the client request. This scenario results in requests taking longer than normal in normal circumstances, and the server stops responding to the requests in extreme cases. As a result, the requests eventually time out.

Low CPU situation is also referred to as freeze, stall, or hang.

The following snapshot shows the output of the **`top`** command in Linux, which shows 0% CPU consumption of a Node.js process:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
61	root	20	0	351276	33888	27052	T	0.0	1.7	0:00.06	node

Figure 11.22: *A Node.js process with no CPU consumption*

Preparation

Change the relevant operating system settings and user limits so that a core dump can be produced in abnormal program conditions, like in the case of a crash.

Configure the application command line so that it produces a diagnostic report upon a signal that is sent to the process when it becomes

unresponsive.

The following snapshot shows how to enable the process to trigger a diagnostic report on demand using a signal issued to it:

```
1. $ node --report-on-signal app.js
```

Symptom

If you are using a single instance of the backend application, the frontend will report the status code 408 or an equivalent error message to indicate that the request has timed out.

If you are using multiple replicas of your application and one of the instances hung, it may not be visible from the frontend as the timed-out request may have been reattempted by one of the other healthy replicas, unless all the replicas hang together.

In some cases, the client will get the response from the server, but it's extremely slow to be of any use and affects the user experience by large.

Useful data

We have learned from the crash section that a core dump is the complete snapshot of the running process, so collecting a core dump on the hung process (without terminating it) is a good idea. In a Linux platform, this is typically achieved with the **`gcore`** command.

Collecting a heap dump is a good idea as well if the application is experiencing high memory demand.

A diagnostic report is a good **First-Failure-Data-Capture** (**FFDC**) document in both cases.

Problem determination

When the process experiences low (or zero CPU), the most probable reason is that the only thread (application thread) is unable to move ahead. Probable reasons for this are listed here:

- The thread is waiting for a lock that is owned by another thread forever

- The thread is waiting for an (wrong) event that is never going to happen
- The thread is waiting on a (wrong) socket that does not accept requests
- The thread is executing (wrong) code that does not terminate

In all the cases, the important step in the debugging process is to check what the main thread is doing while experiencing low CPU.

One of the ways to do this is via Node.js inspector, as follows:

1. Attach the Node.js inspector to the hanging process
2. Inspect the call stack
3. Continue the execution within the inspector
4. Step debug a little to see the bounds of program execution
5. Make inference on the code flow and detect the issue

Another approach is to collect CPU profile of the hanging process that will provide similar information in a slightly different manner. A CPU profile shows the list of methods that were executed in a specific period, with the CPU consumption against each method. This information gives a hint about where the thread has been spending time. This needs to be mapped with the control flow in the application to pinpoint the root cause of low CPU.

The following command can be used to collect CPU profile from the application:

```
1. $ node --prof app.js
```

And the following command can be used to process the generated profile output. Note that the filename of the generated profile output will be different in each case:

```
1. $ node --prof-process isolate-0x53986f70-88-v8.log
```

The following snapshot shows the output of the profiler. A higher number of ticks against a specific function means it has consumed more CPU within the sampling interval:

```
01.     ticks parent  name
02.        67   57.3%  /usr/local/bin/node
03.        53   79.1%    /usr/local/bin/node
04.        30   56.6%      LazyCompile: ~compileForInternalLoader node:internal/bootstrap/loaders:299:27
05.        30  100.0%        LazyCompile: ~nativeModuleRequire node:internal/bootstrap/loaders:332:29
06.         7   23.3%          LazyCompile: ~initializeCJSLoader node:internal/bootstrap/pre_execution:420:29
07.         7  100.0%            LazyCompile: ~prepareMainThreadExecution node:internal/bootstrap/pre_execution:22:36
08.         6   20.0%          Function: ~<anonymous> node:internal/modules/esm/loader:1:1
09.         6  100.0%            LazyCompile: ~compileForInternalLoader node:internal/bootstrap/loaders:299:27
10.         3   10.0%          Function: ~<anonymous> node:internal/modules/esm/get_source:1:1
11.         3  100.0%            LazyCompile: ~compileForInternalLoader node:internal/bootstrap/loaders:299:27
12.         3   10.0%          Function: ~<anonymous> node:internal/main/run_main_module:1:1
13.         2    6.7%          LazyCompile: ~afterInspector node:internal/errors:751:17
```

Figure 11.23: *Sample-based CPU profile of the application*

If the previous two methods are not helping or do not provide sufficient data to inspect, a system dump can be collected on the running process without terminating it.

The following steps demonstrate how to collect a core dump from a running process in Linux:

```
01.  $ ps
02.      PID TTY          TIME CMD
03.    47437 pts/0    00:00:00 bash
04.    47526 pts/0    00:00:00 node
05.    47535 pts/0    00:00:00 ps
06.  $ gcore 47526
07.  [New LWP 47529]
08.  [Thread debugging using libthread_db enabled]
09.  Saved corefile core.47526
10.  [Inferior 1 (process 47526) detached]
11.
12.  $ ls -lrt
13.  total 256908
14.  -rw-r--r-- 1 nodeuser nodeuser 263070256 Aug 21 07:59 core.47526
```

Figure 11.24: *Capturing core dump on a running process*

Collecting and reviewing a diagnostic report is always useful as it enables the native and JavaScript stack to show where the thread is at the moment. Collecting multiple reports back-to-back and analyzing the call stack to determine the progress is also a good idea.

Note: Node.js runtime exposes only one application thread, and those deadlocks are by far the most common hang reasons, so hang-related issues are rare in Node.js workloads. However, given that there are internal helper threads that communicate with the main thread for various reasons, the deadlocks and complete hang are, theoretically, still possible.

Debugging high CPU

High CPU is a scenario when the process is eating up all of its allotted CPU slices without performing any useful work. This is the opposite of low CPU from the processor time consumption point of view, but similar to it from an end result point of view—both make the process hang in an inconsistent state without the ability to deliver any useful work.

Preparation

System preparation for high CPU is exactly same as that for low CPU, that is, change the relevant operating system settings and user limits so that a core dump can be produced upon abnormal program conditions. Additionally, tune the application so that it produces a diagnostic report upon receiving a signal that is sent to the process when it starts looping/spinning.

Symptom

If you are using a single instance of the backend application, the frontend will report the status code 408 or an equivalent error message to indicate that the request is timed out.

If you are using multiple replicas of your application and one of the instances hangs, it may not be visible from the frontend as the timed-out request may be reattempted by other healthy replicas, unless all the replicas hang together.

The following snapshot shows the output of the `top` command in Linux, which shows 100% CPU consumption of a Node.js process:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
152	root	20	0	317468	33992	28448	R	100.0	1.7	0:31.80	node

Figure 11.25: *A Node.js application showing 100% CPU consumption*

Useful data

Just like in the low CPU section, a core dump on the hung process (without terminating it) provides the most comprehensive data. Also, a heap dump,

CPU profile data, and a diagnostic report are good for diagnosing high CPU.

Problem determination

First, the high CPU situation needs to be well qualified. For example, how much is high, and how much is moderate?

The following table shows the average CPU consumption of various workloads running on traditional technologies:

Workload type	Average CPU consumption (%)
Web server	6
Database	9
Email application	7
Application server	6
Web service	7
Node.js web server	80

Table 11.1: *Average CPU consumption of commercial workloads*

Clearly, the consumption is pretty low, say less than 20% on an average, for non-Node.js workloads. The main reason is that every work is a mixture of CPU-bound and I/O-bound operations, with I/O-bound operations taking considerably longer as compared to their CPU-bound counterparts. This means the thread will be seen as blocked for I/O completion for most of the time, resulting in such statistics.

This means anything above 20% is a high CPU for those platforms.

On the other hand, CPU-bound and I/O-bound operations are separated out for Node.js-based applications, and the thread is allowed to run almost all its allotted time, with the absence of any CPU-bound code as the only delimiting factor. This happens when there are no requests in the backlog, at under-load situations.

Under normal circumstances, a Node.js based web application running in production scale load will be seen as consuming as much as 90% of its allocated CPU. This is not a problematic scenario; it only means your application is running at its full potential, leveraging the allocated CPU to

its best possible extent and bringing high level of concurrency to our workload.

This is an important distinction to make, as this understanding will help us identify a real high CPU situation that we need to debug and distinguish it from a normal scenario of our application running smooth.

Many a times, users get suspicious when they look at monitoring tools, see the CPU usage above 80% or so, and compare those values with their understanding of workloads from other technologies.

An easy way to identify a problematic high CPU situation is when the requests are not getting serviced and get timed out instead.

When the process experiences high CPU, the most probable reason is that the only thread (application thread) is doing a lot of work but is unable to move ahead. Probable reasons for such a situation are as follows:

- The thread is executing a (wrong) tight loop with no exit criteria
- The thread is executing a (wrong) wide loop recursively

Note: Tight loop is a small set of instructions and expressions running in a loop with a very large loop counter or a very far loop termination condition. Wide loop is a relatively large set of instructions or expressions, involving multiple code blocks and even functions being executed within a loop, with a large loop counter or a far loop termination condition. The net effect of tight loop and wide loop is full consumption of the allotted CPU, with the program being unable to make move forward.

In both cases, the important step in debugging is to check what the main thread is doing when the process is experiencing high CPU, just as we did for the low CPU case.

One of the ways to do this is via the Node.js inspector, as follows:

1. Attach the Node.js inspector to the hanging process
2. Inspect the call stack
3. Continue the execution within the inspector
4. Step debug to run the program incrementally and determine the bounds of the loop in the program execution
5. Make inference on the code flow and detect the issue

The following command shows how to attach the inspector. First, start the application normally:

$ node app.js

Issue SIGUSR1 signal to the PID of the process to attach the inspector:

```
01.  $ ps -ef | grep node
02.     501 30003 18328   0  4:37PM ttys000    0:00.12 node app.js
03.  $ kill -SIGUSR1 30003
```

Figure 11.26: *Inspecting the process – issuing signal to the process*

The inspector is enabled when we see the following message:

```
01.  $ node app.js
02.  Debugger listening on ws://127.0.0.1:9229/a0c43b22-6d05-4da4-b4a7-a002687b78e3
03.  For help, see: https://nodejs.org/en/docs/inspector
04.  Debugger attached.
```

Figure 11.27: *Inspecting the process – the process enabling the inspector*

At that point, open the code debugger and visualize the call stack, as follows:

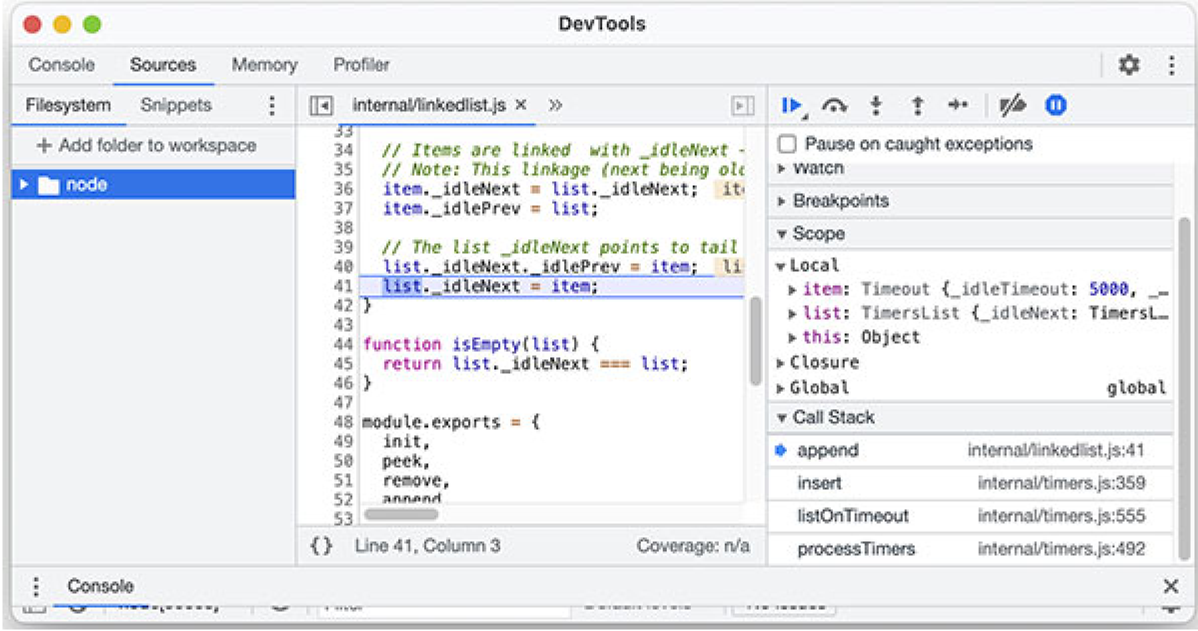

Figure 11.28: *Inspecting the process – various views in Chrome developer tools*

The other approach is to collect CPU profile of the high CPU process that will provide similar information in a slightly different manner. A CPU

profile shows the list of methods that were executed in a specific period, with the CPU consumption against each of the methods recorded separately. This information provides a hint about where the thread has been spending time. This needs to be mapped with the control flow in the application to pinpoint the root cause.

The following snapshot shows the output of performance sampling by the profiler, showing the top level breakup of the CPU:

```
01.      ticks  total  nonlib   name
02.        0     0.0%    0.0%  JavaScript
03.       50    33.8%  100.0%  C++
04.        0     0.0%    0.0%  GC
05.       98    66.2%          Shared libraries
```

Figure 11.29: *Top level break-up of sample CPU profile*

If the preceding two methods do not help or do not provide sufficient data to inspect, a system dump can be collected on the running process without terminating it.

Collecting and reviewing a diagnostic report is always useful, as it enables the native and JavaScript stack to show where the thread is at the moment. Collecting multiple reports back-to-back and analyzing the call stack to determine the progress is also a good idea.

> **Tip:** The sampling profiler output shows the CPU consumption across various dimensions – for easy categorization and abstraction. It shows the breakup in terms of the language (JavaScript / C++ / C), in terms of the threads (application thread and helper threads), and then in terms of the methods.

Debugging memory

Memory anomalies in a web application are of two broad categories:

- Memory leak
- Memory exhaustion

The former is a situation where the program thinks one or more objects as no longer required and not accessed by the program, while the garbage collector finds contrary evidence, leading to the objects lingering around in the program. The latter is an aggravated situation of the former, wherein such objects fill up the entire JavaScript heap and lead to the exhaustion of the heap.

A similar definition holds for native heap memory as well, with the only exception that there is no garbage collection in the native heap. The program itself wrongly assumes the life cycle of some objects and doesn’t free them up.

Preparation

Ensure that the system has sufficient disk space to hold the core dump and the heap dump. A general rule of thumb is to keep a space equal to or greater than double the size of the program’s JavaScript heap size.

Change the relevant operating system settings so that a core dump can be produced upon demand.

Change the relevant user limits on the system so that a core dump can be produced and written to the disk to its fullest.

Install a signal handler in your application that can be invoked upon demand, with a sole function of generating a heap snapshot.

The following code illustrates enabling a signal handler in the application so that heap snapshots can be collected on demand:

```
1. process.on('SIGUSR2', () => {
2. require('v8').writeHeapSnapshot()
3. })
```

Tune the application so that it produces a diagnostic report upon the event of an uncaught exception or an abnormal program termination due to memory exhaustion.

Running the application with the verbose option switched on for garbage collection is also useful while we are in the phase of problem determination.

Symptom

There may not be an identifiable external symptom for small- or medium-scale memory leaks, but large-scale memory leaks can manifest in the form of reduced throughput. On the other hand, a memory exhaustion will crash the server, so the usual rules apply. Some calls will be dropped if the server is running on a single instance, and some calls may be delayed due to the internal retries by the load balancer if there are multiple replicas.

Useful data

A heap dump when the process experiences the memory leak provides the most comprehensive diagnostic data. A core dump and the diagnostic report are good for diagnosing memory leaks and memory exhaustion. Additionally, verbose log of garbage collection history can be very useful at times.

Problem determination

Debugging memory leaks in a well-designed single-threaded web application is relatively easy and straightforward because all the activities in the application are driven by the request-response cycle. In other words, all the objects pertinent to a transaction are created at the time of the request arrival, and all of them are garbage collected (or deemed eligible for garbage collection) when the response has been sent to the wire. This important aspect helps in debugging leaks and also feeds back into the application design; ensure that no/minimal objects are persisted beyond the request-response cycle.

However, this is not as straightforward in practice. For example, a user may want to store some information coming out of every request for auditing purposes. Similarly, they want to cache some responses for certain types of requests and decide which item in the cache matches which request type. They also want to maintain a list of the requests and some identifiable types of those requests. All these lead to additional objects being created on request arrival and escaping the request-response boundary.

The key to memory leak debugging is to precisely know the designed life cycle of the objects that you create in your application.

Debugging memory leak with the help of a heap dump follows this procedure:

1. Locate the top memory retainers in the heap
2. For each retainer, ratify their presence and retention
3. For non-ratified objects, identify (and fix) their life cycle

The following snapshot is the high-level view of the JavaScript heap, with the top level breakup of memory consumption as a chart:

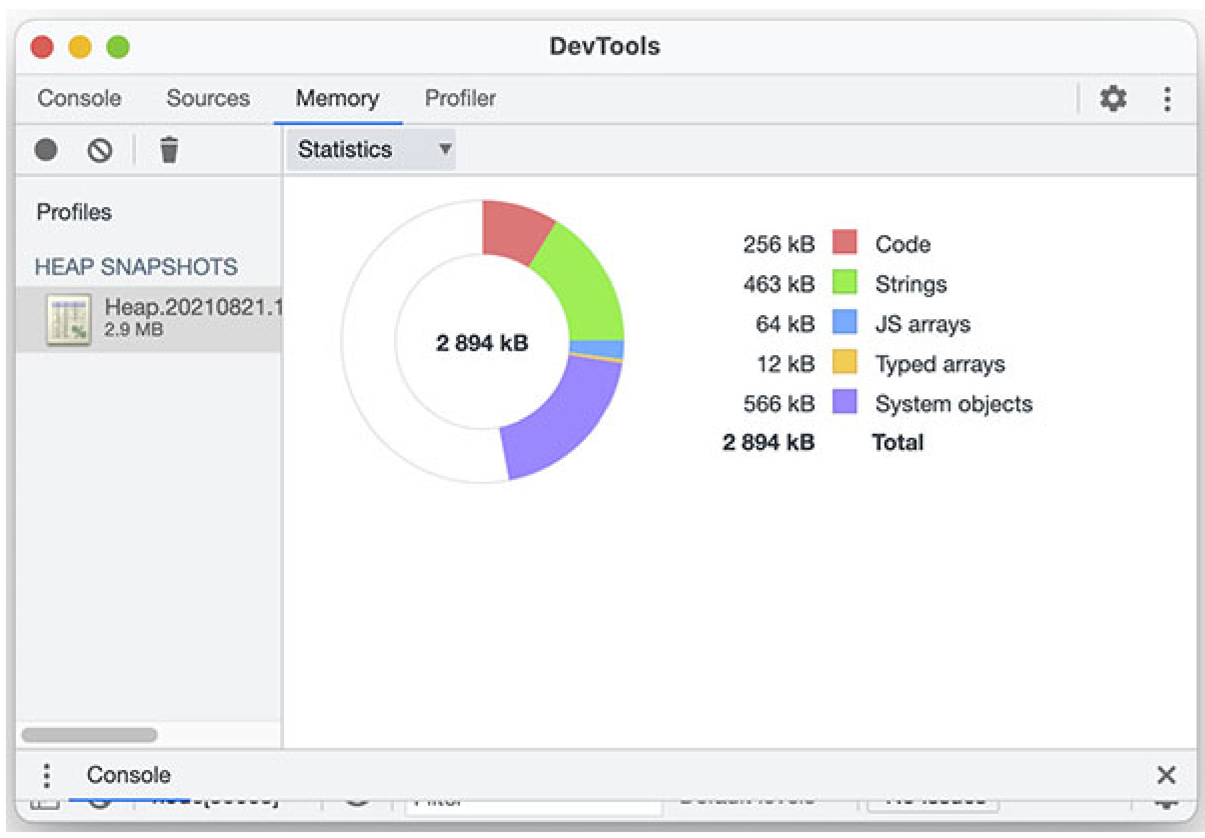

Figure 11.30: *JavaScript heap summary view as a chart*

Locating the top memory retainers is achieved by loading the heap dump in a dump visualizer tool, inspecting the retention statistics, and sorting based on the retained amount.

The following snapshot illustrates the high-level view of the JavaScript heap, with the top level breakup of memory consumption as a list:

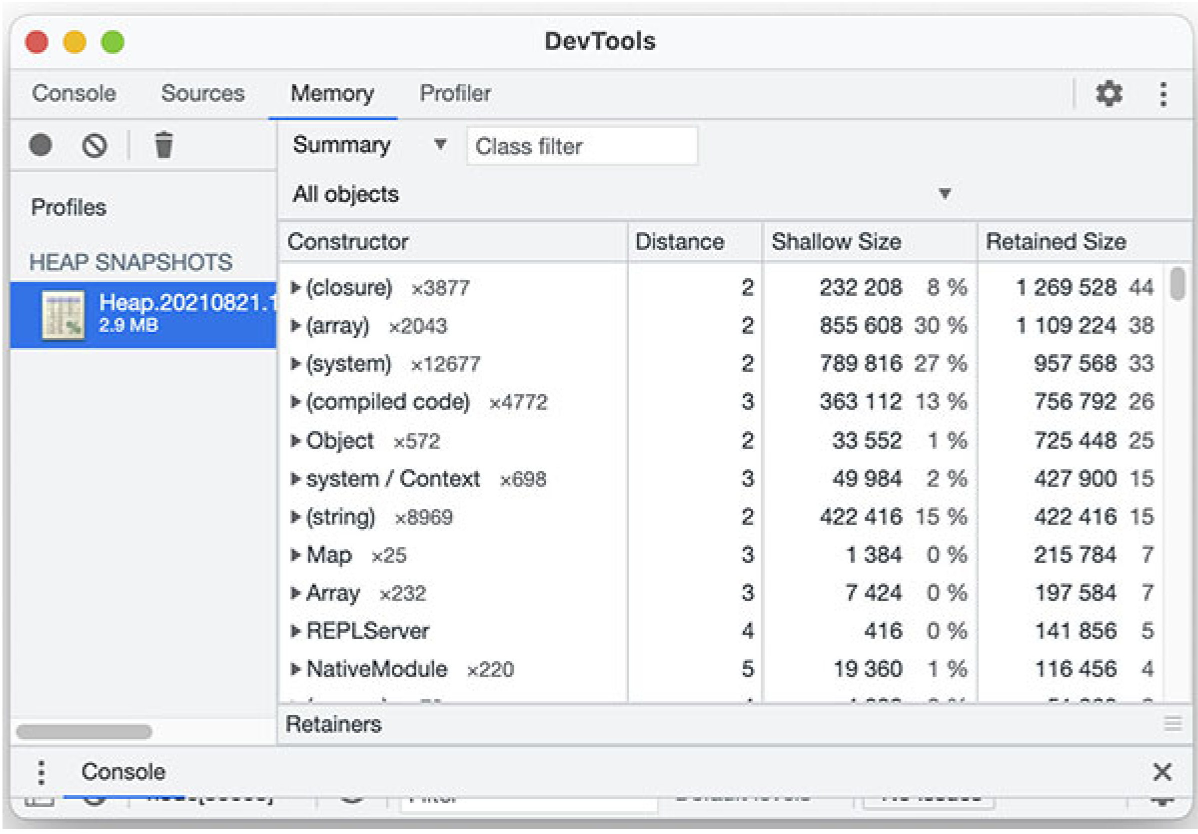

Figure 11.31: *JavaScript heap summary view as a list*

Open the source code and the heap dump side by side to ratify their presence and retention. For each large object shown in the heap dump, review the source code and locate the creation of that object. Make a manual estimate of the size of the object at the time of allocation and its future growth through the code propagation. Check whether the said object being alive at the current control flow point in the application (at the time of the heap dump collection) is valid, and check whether the object carrying the said amount of memory is valid.

If the first validation fails, that is, if we determine that the object should not be alive at this point of time, go to the allocation site of the object, traverse through the code by covering all the control flow points to the current location in the code (at the time of the heap dump collection), and see why the object is still live, or determine which code logic the object is made alive through. Fix the issue by scoping the object appropriately or nullifying the object at the appropriate location in the code.

If the second validation fails, that is, if we determine that the object can be alive at this point of time but its amount of memory retention is

unexpectedly large, perform a similar step as the previous validation, but pay close attention to how the object grows in size. Objects usually grow in size through addition (new fields getting added in an object like structure) or aggregation (new entries appended in a list like structure). Spot the point where unwanted data is getting added to the object and fix it, or spot the point where data is no more valid in the object and purge it.

Note : In the heap explorer, the term shallow size of an object refers to its own size, as defined by its constructor or composition. On the other hand, the retained size refers to the sum total of its own size and the size of all its internal objects, counted cumulatively. In other words, retained size refers to the overall retention volume in the JavaScript heap due to the presence of the said object.

Debugging performance degradation

Performance anomaly is defined as latency or concurrency issues that come in the service invocation, resulting in delayed response to a class of users or overall reduction in the rate of handling requests, leading to poor user experience for the web application.

Preparation

Ensure that the application is started with the "*v8*" sampling profiler when we are debugging for performance issues.

If the application can be modified, it is highly recommended that we apply performance hook APIs that help define measuring points within the request-response cycle and also measure loop latency when we are dealing with highly concurrent workload. This is a vital and unique piece of information on performance debugging.

Symptom

The application is seen as slow in responding, and requests are seen as responded with above-normal latencies, affecting overall user experience. Requests will be seen as timing out under extreme scenarios.

Useful data

There are two classes of useful data that aid performance debugging:

- Data pertinent to the current performance characteristics as a whole
- Complete span data pertinent to an exemplary transaction that is performing poorly

Problem determination

There can be pluralities of techniques for performance debugging: based on various tools, based on various methodologies, and based on various deployment models and execution environments. The one illustrated in the next figure assumes no external tools and does not assume any specific deployment models. So, this is the most generic methodology that will work for any environment, but we will need to specialize this methodology to the specific environment.

General performance debugging follows this procedure:

1. Measure the performance in the entire transaction and break it down
2. Identify the span of transaction where performance is degraded
3. Identify the code blocks that correspond with the transaction span
4. Measure the performance within the span and break it down
5. Repeat this process until the culprit is found and the fix is applied

The following screenshot shows a flame graph visually representing hot methods:

Figure 11.32: A flame graph

The performance in the entire transaction can be measured using the v8 sample profiler, a native CPU sample profiler like Linux perf tool, or the performance counter APIs that are part of the Node.js core APIs.

Identifying the slow span is a manual process by which we enumerate over identifiable sections in the transaction (such as request arrival, HTTP parsing, body composition middleware, database connection, computation, and response composition).

Identifying the code blocks is easy once we can successfully break down the transaction into discrete spans. Just locate the bounds of those spans in the source code.

Note: It is highly recommended to define a performance baseline before you start performance debugging. This means the numbers you measured and ratified when you performed a staging-based performance testing. This helps you precisely find out which area of the transaction and which area of the code are exhibiting outlier behavior in terms of performance, resulting in a better debugging experience.

Measuring the performance within the span depends on the tool we are using. In general, compute the difference between the time of entering and exiting the span, or see if relative CPU consumption against each span is available via sampling.

In some cases, individual transactions may seem alike, and there may be no visible performance issues whatsoever. However, the overall throughput decreases when the number of concurrent requests increase. In this case, the loop latency measurement will be of real help. Use the performance counter API to measure the loop latency and find out how the loop latency increases in response to the number of concurrent connections. Identify the inflection point (the number of concurrent users where the loop latency increases beyond an unacceptable limit) and use that as the concurrency level for the application for the class of requests. We will need to adjust the load balancer or other part of the workload management system to bring this into effect.

Conclusion

In this chapter, we learned problem determination best practices for the most common production anomalies that can occur in a highly concurrent web application.

We learned about the tools, frameworks, methodologies, and best practices for performing basic problem determination of our web application. We also looked at how to understand the type of anomaly by observing the external symptoms of the application and devise a debugging strategy accordingly, choose the right set of tools and methodologies, and troubleshoot the issue ourselves. Many of these are generic learnings that we can apply in other platforms/applications as well. Knowing how to detect, diagnose, and fix web application anomalies wraps up our learning of a web application and its long-term maintenance.

Questions:

1. Why do programs (or libraries or even runtimes) throw assertions or/and exceptions, very well knowing that it can potentially terminate the application?
2. In our application, the JavaScript heap (new space) usage shows a linear increase with the increase in concurrent requests, and at some point, we see objects escape into the old space, which will eventually lead to global garbage collection. Should we adjust the new space size to match the peek concurrency, or should we adjust the admissible concurrent request count to match the current heap setting?
3. The performance (latency) saw a 30% reduction when an application was upgraded to its newer version. How do you pinpoint and resolve the issue?

End

In this book, our aim was to learn about the elements to build an enterprise-grade web application with the help of basic building blocks from the Node.js programming platform. We set an objective of achieving this by understanding all the dimensions of a client-server use case, asking all the relevant questions and answering them ourselves.

We started by looking back on the evolution of languages to meet the need of ever-changing workload characteristics and then discusses Node.js as an exemplary evolution over JavaScript to meet the need of web workloads, which comprise I/O operations. We then made the statement that Node.js is best suited for highly concurrent web workloads due to its peculiar characteristics as a programming model and the runtime architecture. We spent some time understanding event-driven architecture, asynchronous programming, concurrency, parallelism, and scalability, which make up the basic premise of this programming platform.

We then covered the operating space of a web server. It included a variety of components that make up a web application, including the frontend and the backend. We inspected the available APIs in the programming platform required in this space, emphasizing on the networking APIs. Finally, we learned about the enterprise-enabling aspects of a web application to make our application robust and efficient, and we went through the relevant problem determination aspects to debug the most common production issues of web applications.

With that, we believe our learning is complete in all aspects that are required to develop and host an application. We did not focus on any specific tools or frameworks that can change/evolve over time and render the content in this book obsolete. At the same time, we took care not to reinvent the wheel and implement things from ground up. Instead, we attempted to strike a perfect balance between not being biased with any tools while explaining the underlying concepts with a sufficient level of abstraction.

The fundamental concepts and rationales are also discussed when behavior and capabilities are explained, along with asking the most relevant question to invoke an exploratory thought in the reader and solicit architectural interest. This is very important for learners who want to take up Node.js for their prospective career or business. This is because the deployment models and workloads that Node.js hosts are highly dynamic, and developers may constantly be asked to reconsider their program design to meet the ever-changing demands.

We hope this book helps readers understand the implementation details that are required to build an efficient web application based on Node.js CLI, and we hope it is usable as reference material for understanding programming

premises of Node.js and its exemplary use case—web workloads. We sincerely seek feedback on the theme, content, presentation style, or examples and are happy to accommodate those in potentially upcoming editions.

Index

A

accumulator register 73
admin dashboard, website 220
advanced features, website
  admin dashboard 220
  authentication 219, 220
  filters 216, 217
  geolocation indexing 218, 219
  Google APIs 205-209
  Google maps, embedding 210
  pagination 213, 214
  search 214
  search feature, embedding 215, 216
  social media, embedding 211, 212
  user profile 220, 221
application best practice
  API selection 280-283
  application decoupling 277-279
  content optimization 276, 277
  HTTP caching 279, 280
  miscellaneous 283-287
Application Binary Interface (ABI) 73
article
  using 196, 197
asynchronous programming 6
  JavaScript web page example 8
  multimedia example 7, 8
  operating system scheduler example 8, 9
  program interrupt example 6, 7
audio tag
  using 202, 203
authentication 219
availability 231
  issue 232-238
  remediation 232-239
average workload efficiency
  in multi-threaded applications 243
  in single-threaded applications 244

B

backend components
  database 169

external services 165
internal services 158
best practices, Denial of Service
firewalling 155
input validation 155
isolation 155
routing 155
switching 155
best practices, performance 261
application 276
hardware 262
network 264, 265
operating system 265-269
runtime 269
body parser 140
definition 140
implementation 141, 142
use case 140
brute force
remedy 156
threat 155
business logic layer 40

C

Cascading Style Sheets (CSS) 176, 188, 226
client component 39
Closure 25
Cluster module 245, 246
cons 247
pros 247
code editor
selecting 36
components
article 196, 197
audio 202, 203
dialog 197, 198
form 198, 199
hyperlink 195, 196
image 201, 202
options/dropdown 199, 200
script 204, 205
table 200, 201
video 203, 204
concurrency 12
cooking example 12, 13
garbage collection 15
progress bar example 14
concurrency and parallelism
differences 16, 17

similarity 16
concurrency and scalability
horizontal scaling 17, 18
thread pooling 17
cookie 126
attributes 129
definition 127
implementation 127, 128
use case 126
core components, web server 39
business logic layer 40
client 39
microservice layer 41
middleware 40
web layer 39
core features, Node.js
APIs 26
npm 27
small core philosophy 27
streams 27
structure 26
crash
debugging 290-292
Cross-Origin Resource Sharing (CORS) 143
definition 143, 144
implementation 144, 145
use case 143
Cross-Site Scripting (XSS)
remedy 152
threat 152

D

database 169
design considerations 171
intent 170
data privacy and integrity 150
remedy 151
threat 151
debugging crash
performing 290-292
preparation 292, 293
problem determination 294-304
symptom 293
useful data 294
debugging high CPU
preparation 308
problem determination 309-312
symptom 308
useful data 309

debugging low CPU 305
  preparation 305
  problem determination 306-308
  symptom 305
  useful data 306
debugging memory
  preparation 312, 313
  problem determination 313-316
  symptom 313
  useful data 313
debugging performance degradation
  preparation 316
  problem determination 317, 318
  symptom 316
  useful data 316
Denial of Service
  best practices 155
  remedy 155
  threat 154
dependencies
  selecting 35
design considerations, database
  database end 173
  functional 171, 172
  performance 172
  reliability 173
  security 173
  serviceability 174
  web server end 173
design considerations, external services
  functional 167, 168
  performance 168, 169
  reliability 169
  security 169
  serviceability 169
design considerations, internal services 159
  functional 160, 161
  performance 161, 162
  reliability 163
  security 164
  serviceability 164, 165
design considerations, website 177
  articles 179
  consistent pages 183, 184
  ease of use 183
  feedback forms 182
  form 181
  headers and footers 180
  home page 186
  navigation bar 179

- payment options 183
- registered and unregistered views 177, 178
- responsive web forms 186, 187
- search option 181
- shopping carts 182
- short forms 185
- short pages 184, 185
- usability 177

diagnostic report snapshot
- example 255

dialog
- using 197, 198

documentation 259
- best practices 259, 260

Document Object Model (DOM) 189, 190

dumping 252

dumps
- head dump 253
- system dump 253
- textual dump 254

dynamic web page 145
- definition 145, 146
- implementation 146-148
- use case 145

E

elements
- DOM 189
- WebSocket 193, 194
- WebWorker 195
- XMLHttpRequest 190-193

endpoints 125, 126

enterprise enabling components 224
- availability 231
- documentation 259
- issue 228-231
- observability 249
- performance 224
- reliability 225
- remediation 226-231
- scalability 239
- security 256

environment setup
- code editor selection 36
- dependencies selection 35
- Node.js version selection 30, 31
- platform selection 30
- resource requirements 35

event-driven architecture 2

calculator example 2
echo example 3, 4
messaging example 5
number sorting example 2, 3
web page access example 4
web server example 4, 5
Windows event loop 5
event loop 24, 26
extensibility 52
best practices 53, 54
examples 52, 53
external services, backend components 165
design considerations 167
intent 165, 166

F

filter controls
applying 216, 217
filter functions 84
First-Failure-Data-Capture (FFDC) 294
form
using 198, 199
functional programming 9
Closure context 12
Closure functions 10, 11
function as assignment subject 10
function passed as value 9
function returned as result 10

G

geolocation indexing 218
using 218, 219
Google APIs 205-209
Google maps
embedding, into web page 210

H

hardware best practice
cache 263, 264
CPU 262, 263
disk 264
'hello world' program 60
accessing through browser 60, 61
running 60
high availability
cluster and load balancer 236
master-slave replication 235

peer-peer replication 235, 236
high CPU
debugging 308
horizontal scalability 248
example 248, 249
HTTP 2 107
HTTPS 107
HTTP status codes 148
definition 149
implementation 149
use case 149
HTTP verbs 118, 121
definition 119
implementation 119-122
use case 118
hyperlink
using 195, 196
Hyper Text Markup Language (HTML) 187
Hyper Text Transfer Protocol (HTTP) 77
sample form data 78
sample HTTP payload 79
tokens 79
visualization of contract 79

I

image tag
using 201, 202
internal services, backend components
design considerations 159
intent 158, 159
International Standards Organization (ISO) 75

J

JavaScript 188, 189
JavaScript editors
Atom 36
notepad 36
Sublime Text 36
vi editor 36
Visual Studio Code 36

L

languages
CSS 188
HTML 187, 188
JavaScript 188, 189
load balancer 248

Long-term Supported (LTS) 30
low CPU
  debugging 305

M

maintainability 54
  best practices 54, 55
  importance 54
memory
  debugging 312
microservice architecture
  simple representation 45
microservice layer 41
middleware component 40
multipart form-data 138
  definition 138
  implementation 139, 140
  use case 138
multiple Node.js processes 244, 245
  cons 245
  pros 245

N

Natural Language Processing (NLP) 84
network best practice 264, 265
networking APIs 107
  HTTP 2 107
  HTTPS 107
  User Datagram Protocol (UDP) 107
network programming 70
Node.js
  asynchronous programming 21, 22
  core features 26
  CPU bound operations 19
  event-driven architecture 22-24
  functional programming 25, 26
  installing 31-35
  I/O bound operations 19
  latency in computer systems 18, 19
  multiple Node.js processes 244
  multi-thread view 242, 243
  program sections 64
  scalability 20, 21
  URL 31
  version, selecting 30, 31
  web workload characteristics 19, 20
Node.js buffers 86-91
Node.js module 245-247

Node.js streams 82-85
Node Package Manager 27
npm modules 27

O

observability 56, 249
  best practices 57
  dumping 252-255
  importance 56
  key elements 249
  monitoring 252
  profiling 251
  trace 250, 251
Open Systems Interconnection (OSI) 75
operating system best practice 265-269
option tag
  usage 199, 200
OSI data interchange architecture
  layers 75

P

pagination 213
  implementation 213, 214
parallelism 15
  distributed word counting example 16
performance 224
performance degradation
  debugging 316
platforms
  selecting 30
program anomalies
  crash, debugging 290
  debugging 289
  high CPU, debugging 308
  low CPU, debugging 305
  memory, debugging 312
  performance degradation, debugging 316
program sections, Node.js
  date 65
  require statement 64
  server listen 66
  server loop 64, 65
  server response 66
progress bar 14
protocol 71
  at called function side 71
  at calling function side 71
  simple function call example 71-73

Q

query injection
  remedy 154
  threat 153

R

RAS) features 223
reliability 50, 225, 226
  best practices 51, 52
  issues 50, 51, 226
replicas 248
request forwarding 133
  definition 133, 134
  implementation 134-137
  use case 133
request life cycle 96
  aborted 96-98
  close 97
  data 97
  end 98
request object 91-93
require statement 64
resource requirements
  selecting 35
response life cycle
  close 99
  finish 99, 100
response object 94-96
route 122
  definition 122, 123
  implementation 123, 124
  use case 122
runtime best practice 269
  concurrency 274
  concurrency, increasing 274-276
  debug options, removing 276
  event loop 272, 273
  garbage collection 269-272

S

scalability 239, 240
  horizontal scalability 248, 249
  vertical scalability 240, 241
script tag
  using 204, 205
search engine 214
  embedding, into web page 215

security 256
  authentication 257
  authorization 257
  data encryption 258
  file system protection 258, 259
  headers 256, 257
  input validation 256
  logging, auditing 258
  sessions 257
  unwanted ports, shutting down 259
security issues, server security
  brute force 155
  Cross-Site Scripting (XSS) 152
  data privacy and integrity 150
  Denial of Service 154
  query injection 153
security measures
  connection level 174
  database level 174
  query level 174
  transport level 174
server capabilities
  incremental buildup 110
server configuration 100, 101
  maxHeadersCount 101, 102
  timecount 102, 103
server listen 66
server loop 64
server response 66
server security 150
  definition 150
  implementation 150
  use case 150
server's life cycle events 103
  close 106
  connect 105
  listening 103-105
serviceability 55
  best practices 56
Service Level Agreements (SLAs) 52
Service Level Objective (SLOs) 52
session 129
  definition 129, 130
  implementation 130-133
  use case 129
SetNoDelay API 265
sharding 238
snapshotting 252
social media share
  embedding, into web page 211, 212

static file server 112
definition 112
implementation 113, 114
limitations 115-118
use case 112
stdout log 294
Symmetric Multi-Threading (SMT) 264

T

table
using 200, 201
TCP-based Node.js server
example 76, 77
TCP/IP 74
graphical view 75
programming interfaces 76
TCP/IP layers
application 76
Internet 76
physical 76
transport 76
Thread 17
thread pool 17
Time of the Day server 61
accessing through browser 62, 63
tokens, HTTP
content-length 81
content-type 80
Host 80
HTTP/1.1 80
POST 79, 80
/ verb 80
tracing 250
common practices 251

U

Universal Co-ordinated Time (UTC) 62
User Datagram Protocol (UDP) 107
user profile page 220, 221

V

vertical scalability
issue 241
remediation 242
video tag
using 203, 204

W

web application
  reliability matrix 225
web layer component 39
web pages
  history 176
web server 38
  architecture 38, 39
  comparing with desktop 42, 43
  considerations 41
  core components 39
web server considerations
  architecture 44
  extensibility 52
  extensibility best practices 53, 54
  extensibility examples 52, 53
  maintainability 54
  maintainability best practices 54, 55
  microservice architecture 44, 45
  observability 56
  observability best practices 57
  performance 45, 46
  performance best practices 47
  performance overheads 46, 47
  reliability 50
  reliability best practices 51, 52
  reliability issues, examples 50, 51
  security 48
  security best practices 49, 50
  security threat types 48
  serviceability 55
  serviceability best practices 56
website
  advanced features 205
  components 195
  design considerations 177
  elements 189
  elements and components 187
  languages 187
WebSocket 193, 194
WebWorker 195
win32 applications 5
worker threads module 247, 248
  cons 248
  pros 248

X

XMLHttpRequest 190-193