

# Node.js Design Patterns

*Third Edition*

Design and implement production-grade Node.js applications using proven patterns and techniques

**Mario Casciaro**

**Luciano Mammino**



BIRMINGHAM - MUMBAI

# Node.js Design Patterns

*Third Edition*

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Producer:** Tushar Gupta

**Acquisition Editor – Peer Reviews:** Suresh Jain

**Project Editor:** Tom Jacob

**Content Development Editors:** Joanne Lovell, Bhavesh Amin

**Copy Editor:** Safis Editing

**Technical Editor:** Saby D'silva

**Proofreader:** Safis Editing

**Indexer:** Manju Arasan

**Presentation Designer:** Sandip Tadge

First published: December 2014

Second Edition: July 2016

Third Edition: July 2020

Production reference: 1240720

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-83921-411-0

[www.packt.com](http://www.packt.com)

# Table of Contents

|                                                 |           |
|-------------------------------------------------|-----------|
| <b>Preface</b>                                  | <b>xi</b> |
| <b>Chapter 1: The Node.js Platform</b>          | <b>1</b>  |
| <b>The Node.js philosophy</b>                   | <b>2</b>  |
| Small core                                      | 2         |
| Small modules                                   | 2         |
| Small surface area                              | 3         |
| Simplicity and pragmatism                       | 4         |
| <b>How Node.js works</b>                        | <b>5</b>  |
| I/O is slow                                     | 5         |
| Blocking I/O                                    | 5         |
| Non-blocking I/O                                | 6         |
| Event demultiplexing                            | 7         |
| The reactor pattern                             | 9         |
| Libuv, the I/O engine of Node.js                | 11        |
| The recipe for Node.js                          | 12        |
| <b>JavaScript in Node.js</b>                    | <b>13</b> |
| Run the latest JavaScript with confidence       | 13        |
| The module system                               | 14        |
| Full access to operating system services        | 14        |
| Running native code                             | 15        |
| <b>Summary</b>                                  | <b>16</b> |
| <b>Chapter 2: The Module System</b>             | <b>17</b> |
| <b>The need for modules</b>                     | <b>18</b> |
| <b>Module systems in JavaScript and Node.js</b> | <b>19</b> |
| <b>The module system and its patterns</b>       | <b>20</b> |
| The revealing module pattern                    | 20        |

|                                                          |           |
|----------------------------------------------------------|-----------|
| <b>CommonJS modules</b>                                  | <b>22</b> |
| A homemade module loader                                 | 22        |
| Defining a module                                        | 24        |
| module.exports versus exports                            | 25        |
| The require function is synchronous                      | 26        |
| The resolving algorithm                                  | 26        |
| The module cache                                         | 28        |
| Circular dependencies                                    | 29        |
| <b>Module definition patterns</b>                        | <b>33</b> |
| Named exports                                            | 33        |
| Exporting a function                                     | 34        |
| Exporting a class                                        | 35        |
| Exporting an instance                                    | 36        |
| Modifying other modules or the global scope              | 37        |
| <b>ESM: ECMAScript modules</b>                           | <b>38</b> |
| Using ESM in Node.js                                     | 39        |
| Named exports and imports                                | 39        |
| Default exports and imports                              | 42        |
| Mixed exports                                            | 43        |
| Module identifiers                                       | 45        |
| Async imports                                            | 45        |
| Module loading in depth                                  | 48        |
| Loading phases                                           | 48        |
| Read-only live bindings                                  | 49        |
| Circular dependency resolution                           | 50        |
| Modifying other modules                                  | 56        |
| <b>ESM and CommonJS differences and interoperability</b> | <b>60</b> |
| ESM runs in strict mode                                  | 60        |
| Missing references in ESM                                | 60        |
| Interoperability                                         | 61        |
| <b>Summary</b>                                           | <b>62</b> |
| <b>Chapter 3: Callbacks and Events</b>                   | <b>63</b> |
| <b>The Callback pattern</b>                              | <b>64</b> |
| The continuation-passing style                           | 64        |
| Synchronous CPS                                          | 65        |
| Asynchronous CPS                                         | 65        |
| Non-CPS callbacks                                        | 67        |
| Synchronous or asynchronous?                             | 67        |
| An unpredictable function                                | 68        |
| Unleashing Zalgo                                         | 68        |
| Using synchronous APIs                                   | 70        |
| Guaranteeing asynchronicity with deferred execution      | 72        |

---

|                                                                                    |            |
|------------------------------------------------------------------------------------|------------|
| Node.js callback conventions                                                       | 73         |
| The callback comes last                                                            | 73         |
| Any error always comes first                                                       | 74         |
| Propagating errors                                                                 | 74         |
| Uncaught exceptions                                                                | 75         |
| <b>The Observer pattern</b>                                                        | <b>77</b>  |
| The EventEmitter                                                                   | 78         |
| Creating and using the EventEmitter                                                | 79         |
| Propagating errors                                                                 | 80         |
| Making any object observable                                                       | 80         |
| EventEmitter and memory leaks                                                      | 82         |
| Synchronous and asynchronous events                                                | 83         |
| EventEmitter versus callbacks                                                      | 85         |
| Combining callbacks and events                                                     | 86         |
| <b>Summary</b>                                                                     | <b>88</b>  |
| <b>Exercises</b>                                                                   | <b>88</b>  |
| <b>Chapter 4: Asynchronous Control Flow Patterns with Callbacks</b>                | <b>89</b>  |
| <hr/>                                                                              |            |
| <b>The difficulties of asynchronous programming</b>                                | <b>90</b>  |
| Creating a simple web spider                                                       | 90         |
| Callback hell                                                                      | 93         |
| <b>Callback best practices and control flow patterns</b>                           | <b>94</b>  |
| Callback discipline                                                                | 95         |
| Applying the callback discipline                                                   | 95         |
| Sequential execution                                                               | 98         |
| Executing a known set of tasks in sequence                                         | 99         |
| Sequential iteration                                                               | 100        |
| Parallel execution                                                                 | 104        |
| Web spider version 3                                                               | 106        |
| The pattern                                                                        | 108        |
| Fixing race conditions with concurrent tasks                                       | 108        |
| Limited parallel execution                                                         | 110        |
| Limiting concurrency                                                               | 112        |
| Globally limiting concurrency                                                      | 113        |
| <b>The async library</b>                                                           | <b>119</b> |
| <b>Summary</b>                                                                     | <b>120</b> |
| <b>Exercises</b>                                                                   | <b>121</b> |
| <b>Chapter 5: Asynchronous Control Flow Patterns with Promises and Async/Await</b> | <b>123</b> |
| <hr/>                                                                              |            |
| <b>Promises</b>                                                                    | <b>124</b> |
| What is a promise?                                                                 | 125        |
| Promises/A+ and thenables                                                          | 127        |
| The promise API                                                                    | 128        |

|                                                                         |            |
|-------------------------------------------------------------------------|------------|
| Creating a promise                                                      | 130        |
| Promisification                                                         | 131        |
| Sequential execution and iteration                                      | 133        |
| Parallel execution                                                      | 136        |
| Limited parallel execution                                              | 137        |
| Implementing the TaskQueue class with promises                          | 138        |
| Updating the web spider                                                 | 139        |
| <b>Async/await</b>                                                      | <b>141</b> |
| Async functions and the await expression                                | 141        |
| Error handling with async/await                                         | 143        |
| A unified try...catch experience                                        | 143        |
| The "return" versus "return await" trap                                 | 144        |
| Sequential execution and iteration                                      | 145        |
| Antipattern – using async/await with Array.forEach for serial execution | 147        |
| Parallel execution                                                      | 147        |
| Limited parallel execution                                              | 149        |
| <b>The problem with infinite recursive promise resolution chains</b>    | <b>152</b> |
| <b>Summary</b>                                                          | <b>156</b> |
| <b>Exercises</b>                                                        | <b>157</b> |
| <b>Chapter 6: Coding with Streams</b>                                   | <b>159</b> |
| <b>Discovering the importance of streams</b>                            | <b>160</b> |
| Buffering versus streaming                                              | 160        |
| Spatial efficiency                                                      | 161        |
| Gzipping using a buffered API                                           | 162        |
| Gzipping using streams                                                  | 163        |
| Time efficiency                                                         | 163        |
| Composability                                                           | 167        |
| Adding client-side encryption                                           | 167        |
| Adding server-side decryption                                           | 169        |
| <b>Getting started with streams</b>                                     | <b>170</b> |
| Anatomy of streams                                                      | 170        |
| Readable streams                                                        | 171        |
| Reading from a stream                                                   | 171        |
| Implementing Readable streams                                           | 174        |
| Writable streams                                                        | 179        |
| Writing to a stream                                                     | 179        |
| Backpressure                                                            | 181        |
| Implementing Writable streams                                           | 182        |
| Duplex streams                                                          | 185        |
| Transform streams                                                       | 185        |
| Implementing Transform streams                                          | 186        |
| Filtering and aggregating data with Transform streams                   | 189        |
| PassThrough streams                                                     | 193        |
| Observability                                                           | 193        |

|                                                        |            |
|--------------------------------------------------------|------------|
| Late piping                                            | 194        |
| Lazy streams                                           | 197        |
| Connecting streams using pipes                         | 198        |
| Pipes and error handling                               | 200        |
| Better error handling with pipeline()                  | 201        |
| <b>Asynchronous control flow patterns with streams</b> | <b>203</b> |
| Sequential execution                                   | 203        |
| Unordered parallel execution                           | 206        |
| Implementing an unordered parallel stream              | 206        |
| Implementing a URL status monitoring application       | 208        |
| Unordered limited parallel execution                   | 210        |
| Ordered parallel execution                             | 212        |
| <b>Piping patterns</b>                                 | <b>214</b> |
| Combining streams                                      | 214        |
| Implementing a combined stream                         | 217        |
| Forking streams                                        | 219        |
| Implementing a multiple checksum generator             | 220        |
| Merging streams                                        | 221        |
| Merging text files                                     | 221        |
| Multiplexing and demultiplexing                        | 223        |
| Building a remote logger                               | 224        |
| Multiplexing and demultiplexing object streams         | 229        |
| <b>Summary</b>                                         | <b>230</b> |
| <b>Exercises</b>                                       | <b>230</b> |
| <b>Chapter 7: Creational Design Patterns</b>           | <b>233</b> |
| <b>Factory</b>                                         | <b>234</b> |
| Decoupling object creation and implementation          | 235        |
| A mechanism to enforce encapsulation                   | 236        |
| Building a simple code profiler                        | 238        |
| In the wild                                            | 241        |
| <b>Builder</b>                                         | <b>241</b> |
| Implementing a URL object builder                      | 244        |
| In the wild                                            | 248        |
| <b>Revealing Constructor</b>                           | <b>249</b> |
| Building an immutable buffer                           | 250        |
| In the wild                                            | 253        |
| <b>Singleton</b>                                       | <b>253</b> |
| <b>Wiring modules</b>                                  | <b>257</b> |
| Singleton dependencies                                 | 258        |
| Dependency Injection                                   | 261        |
| <b>Summary</b>                                         | <b>266</b> |
| <b>Exercises</b>                                       | <b>267</b> |

|                                                          |            |
|----------------------------------------------------------|------------|
| <b>Chapter 8: Structural Design Patterns</b>             | <b>269</b> |
| <b>Proxy</b>                                             | <b>269</b> |
| Techniques for implementing proxies                      | 271        |
| Object composition                                       | 272        |
| Object augmentation                                      | 275        |
| The built-in Proxy object                                | 277        |
| A comparison of the different proxying techniques        | 280        |
| Creating a logging Writable stream                       | 281        |
| Change observer with Proxy                               | 282        |
| In the wild                                              | 285        |
| <b>Decorator</b>                                         | <b>285</b> |
| Techniques for implementing decorators                   | 286        |
| Composition                                              | 286        |
| Object augmentation                                      | 288        |
| Decorating with the Proxy object                         | 289        |
| Decorating a LevelUP database                            | 290        |
| Introducing LevelUP and LevelDB                          | 290        |
| Implementing a LevelUP plugin                            | 291        |
| In the wild                                              | 293        |
| <b>The line between proxy and decorator</b>              | <b>294</b> |
| <b>Adapter</b>                                           | <b>294</b> |
| Using LevelUP through the filesystem API                 | 295        |
| In the wild                                              | 298        |
| <b>Summary</b>                                           | <b>299</b> |
| <b>Exercises</b>                                         | <b>300</b> |
| <b>Chapter 9: Behavioral Design Patterns</b>             | <b>301</b> |
| <b>Strategy</b>                                          | <b>302</b> |
| Multi-format configuration objects                       | 304        |
| In the wild                                              | 308        |
| <b>State</b>                                             | <b>308</b> |
| Implementing a basic failsafe socket                     | 310        |
| <b>Template</b>                                          | <b>315</b> |
| A configuration manager template                         | 316        |
| In the wild                                              | 318        |
| <b>Iterator</b>                                          | <b>319</b> |
| The iterator protocol                                    | 319        |
| The iterable protocol                                    | 322        |
| Iterators and iterables as a native JavaScript interface | 324        |
| Generators                                               | 326        |
| Generators in theory                                     | 327        |
| A simple generator function                              | 327        |
| Controlling a generator iterator                         | 328        |
| How to use generators in place of iterators              | 330        |



|                                                              |            |
|--------------------------------------------------------------|------------|
| Async iterators                                              | 331        |
| Async generators                                             | 334        |
| Async iterators and Node.js streams                          | 335        |
| In the wild                                                  | 336        |
| <b>Middleware</b>                                            | <b>337</b> |
| Middleware in Express                                        | 337        |
| Middleware as a pattern                                      | 338        |
| Creating a middleware framework for ZeroMQ                   | 340        |
| The Middleware Manager                                       | 340        |
| Implementing the middleware to process messages              | 342        |
| Using the ZeroMQ middleware framework                        | 344        |
| In the wild                                                  | 347        |
| <b>Command</b>                                               | <b>347</b> |
| The Task pattern                                             | 349        |
| A more complex command                                       | 349        |
| <b>Summary</b>                                               | <b>353</b> |
| <b>Exercises</b>                                             | <b>354</b> |
| <b>Chapter 10: Universal JavaScript for Web Applications</b> | <b>357</b> |
| <b>Sharing code with the browser</b>                         | <b>358</b> |
| JavaScript modules in a cross-platform context               | 359        |
| Module bundlers                                              | 360        |
| How a module bundler works                                   | 363        |
| Using webpack                                                | 369        |
| <b>Fundamentals of cross-platform development</b>            | <b>371</b> |
| Runtime code branching                                       | 372        |
| Challenges of runtime code branching                         | 373        |
| Build-time code branching                                    | 374        |
| Module swapping                                              | 377        |
| Design patterns for cross-platform development               | 378        |
| <b>A brief introduction to React</b>                         | <b>379</b> |
| Hello React                                                  | 381        |
| Alternatives to react.createElement                          | 383        |
| Stateful components                                          | 385        |
| <b>Creating a Universal JavaScript app</b>                   | <b>391</b> |
| Frontend-only app                                            | 392        |
| Server-side rendering                                        | 399        |
| Asynchronous data retrieval                                  | 405        |
| Universal data retrieval                                     | 411        |
| Two-pass rendering                                           | 412        |
| Async pages                                                  | 414        |
| Implementing async pages                                     | 416        |
| <b>Summary</b>                                               | <b>425</b> |
| <b>Exercises</b>                                             | <b>426</b> |

|                                                           |            |
|-----------------------------------------------------------|------------|
| <b>Chapter 11: Advanced Recipes</b>                       | <b>427</b> |
| <b>Dealing with asynchronously initialized components</b> | <b>428</b> |
| The issue with asynchronously initialized components      | 428        |
| Local initialization check                                | 429        |
| Delayed startup                                           | 430        |
| Pre-initialization queues                                 | 431        |
| In the wild                                               | 435        |
| <b>Asynchronous request batching and caching</b>          | <b>435</b> |
| What's asynchronous request batching?                     | 436        |
| Optimal asynchronous request caching                      | 437        |
| An API server without caching or batching                 | 439        |
| Batching and caching with promises                        | 441        |
| Batching requests in the total sales web server           | 442        |
| Caching requests in the total sales web server            | 443        |
| Notes about implementing caching mechanisms               | 445        |
| <b>Canceling asynchronous operations</b>                  | <b>445</b> |
| A basic recipe for creating cancelable functions          | 446        |
| Wrapping asynchronous invocations                         | 447        |
| Cancelable async functions with generators                | 449        |
| <b>Running CPU-bound tasks</b>                            | <b>453</b> |
| Solving the subset sum problem                            | 453        |
| Interleaving with setImmediate                            | 457        |
| Interleaving the steps of the subset sum algorithm        | 457        |
| Considerations on the interleaving approach               | 459        |
| Using external processes                                  | 460        |
| Delegating the subset sum task to an external process     | 461        |
| Considerations for the multi-process approach             | 467        |
| Using worker threads                                      | 468        |
| Running the subset sum task in a worker thread            | 469        |
| Running CPU-bound tasks in production                     | 472        |
| <b>Summary</b>                                            | <b>473</b> |
| <b>Exercises</b>                                          | <b>473</b> |
| <b>Chapter 12: Scalability and Architectural Patterns</b> | <b>475</b> |
| <b>An introduction to application scaling</b>             | <b>476</b> |
| Scaling Node.js applications                              | 477        |
| The three dimensions of scalability                       | 477        |
| <b>Cloning and load balancing</b>                         | <b>479</b> |
| The cluster module                                        | 480        |
| Notes on the behavior of the cluster module               | 481        |
| Building a simple HTTP server                             | 482        |
| Scaling with the cluster module                           | 484        |
| Resiliency and availability with the cluster module       | 486        |
| Zero-downtime restart                                     | 488        |

|                                                                               |            |
|-------------------------------------------------------------------------------|------------|
| Dealing with stateful communications                                          | 490        |
| Sharing the state across multiple instances                                   | 491        |
| Sticky load balancing                                                         | 492        |
| Scaling with a reverse proxy                                                  | 494        |
| Load balancing with Nginx                                                     | 496        |
| Dynamic horizontal scaling                                                    | 501        |
| Using a service registry                                                      | 501        |
| Implementing a dynamic load balancer with http-proxy and Consul               | 503        |
| Peer-to-peer load balancing                                                   | 510        |
| Implementing an HTTP client that can balance requests across multiple servers | 511        |
| Scaling applications using containers                                         | 513        |
| What is a container?                                                          | 513        |
| Creating and running a container with Docker                                  | 514        |
| What is Kubernetes?                                                           | 517        |
| Deploying and scaling an application on Kubernetes                            | 519        |
| <b>Decomposing complex applications</b>                                       | <b>523</b> |
| Monolithic architecture                                                       | 524        |
| The microservice architecture                                                 | 526        |
| An example of a microservice architecture                                     | 526        |
| Microservices – advantages and disadvantages                                  | 528        |
| Integration patterns in a microservice architecture                           | 530        |
| The API proxy                                                                 | 531        |
| API orchestration                                                             | 532        |
| Integration with a message broker                                             | 536        |
| <b>Summary</b>                                                                | <b>538</b> |
| <b>Exercises</b>                                                              | <b>539</b> |
| <b>Chapter 13: Messaging and Integration Patterns</b>                         | <b>541</b> |
| <b>Fundamentals of a messaging system</b>                                     | <b>542</b> |
| One way versus request/reply patterns                                         | 542        |
| Message types                                                                 | 544        |
| Command Messages                                                              | 544        |
| Event Messages                                                                | 545        |
| Document Messages                                                             | 545        |
| Asynchronous messaging, queues, and streams                                   | 545        |
| Peer-to-peer or broker-based messaging                                        | 547        |
| <b>Publish/Subscribe pattern</b>                                              | <b>549</b> |
| Building a minimalist real-time chat application                              | 550        |
| Implementing the server side                                                  | 550        |
| Implementing the client side                                                  | 551        |
| Running and scaling the chat application                                      | 553        |
| Using Redis as a simple message broker                                        | 554        |
| Peer-to-peer Publish/Subscribe with ZeroMQ                                    | 557        |
| Introducing ZeroMQ                                                            | 557        |
| Designing a peer-to-peer architecture for the chat server                     | 558        |
| Using the ZeroMQ PUB/SUB sockets                                              | 559        |

|                                                                        |            |
|------------------------------------------------------------------------|------------|
| Reliable message delivery with queues                                  | 562        |
| Introducing AMQP                                                       | 564        |
| Durable subscribers with AMQP and RabbitMQ                             | 566        |
| Reliable messaging with streams                                        | 571        |
| Characteristics of a streaming platform                                | 571        |
| Streams versus message queues                                          | 573        |
| Implementing the chat application using Redis Streams                  | 573        |
| <b>Task distribution patterns</b>                                      | <b>577</b> |
| The ZeroMQ Fanout/Fanin pattern                                        | 579        |
| PUSH/PULL sockets                                                      | 580        |
| Building a distributed hashsum cracker with ZeroMQ                     | 580        |
| Pipelines and competing consumers in AMQP                              | 587        |
| Point-to-point communications and competing consumers                  | 588        |
| Implementing the hashsum cracker using AMQP                            | 588        |
| Distributing tasks with Redis Streams                                  | 592        |
| Redis consumer groups                                                  | 593        |
| Implementing the hashsum cracker using Redis Streams                   | 594        |
| <b>Request/Reply patterns</b>                                          | <b>598</b> |
| Correlation Identifier                                                 | 598        |
| Implementing a request/reply abstraction using correlation identifiers | 599        |
| Return address                                                         | 605        |
| Implementing the Return Address pattern in AMQP                        | 605        |
| <b>Summary</b>                                                         | <b>611</b> |
| <b>Exercises</b>                                                       | <b>612</b> |
| <b>Other Books You May Enjoy</b>                                       | <b>615</b> |
| <b>Index</b>                                                           | <b>619</b> |

---