OKIEM EKSPERTA

Node.js

Wzorce projektowe i techniki tworzenia aplikacji produkcyjnych

Wydanie IV

Wstępy napisali:
Colin J. Ihrig (współtwórca Node.js)
Matteo Collina (główny opiekun Fastify, współtwórca Node.js)

Luciano Mammino
Mario Casciaro

Tytuł oryginału: Node.js Design Patterns: Level up your Node.js skills and design production-grade applications using proven techniques, 4th Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-3647-8

Copyright © Packt Publishing 2025.

First published in the English language under the title
'Node.js Design Patterns - Fourth Edition – (9781803238944)'

Polish edition copyright © 2026 by Helion S.A.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/nodew4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorach 13

O korektorach merytorycznych 14

Beta czytelnicy 14

Wstęp 1 15

Wstęp 2 17

Wprowadzenie 19

ROZDZIAŁ 1
Platforma Node.js 26

- Filozofia Node.js 27
  - Mały rdzeń 28
  - Małe moduły 28
  - Mała powierzchnia 30
  - Prostota i pragmatyzm 30
- Jak działa Node.js? 31
  - Wąskim gardłem są często operacje wejścia-wyjścia 31
  - Blokujące operacje wejścia-wyjścia 32
  - Nieblokujące operacje wejścia-wyjścia 33
  - Demultipleksacja zdarzeń 34
  - Wzorzec reaktora 36
  - libuv, czyli silnik wejścia-wyjścia w Node.js 38
  - Kompletna receptura dla Node.js 38
- JavaScript w Node.js 39
  - Bez obaw uruchamiaj najnowszy kod napisany w JavaScripcie 40
  - System modułowy 41
  - Pełny dostęp do usług systemu operacyjnego 42
  - Uruchamianie kodu natywnego 42
  - Node.js i TypeScript 43
- Podsumowanie 46

Kup ksi k

Pole ksi k

ROZDZIAŁ 2

System modułów 47
Zalety stosowania modułów 48
Systemy modułów w JavaScripcie i w Node.js 48
Wzorzec ujawnianego modułu 49
Moduły ES 51
Korzystanie z modułów ES w Node.js 52
Składnia modułów ES 53
Algorytm rozwiązywania modułów 61
Szczegółowe omówienie procesu wczytywania modułów 63
Moduły modyfikujące inne moduły 71
Moduły CommonJS 78
Moduły ES i CommonJS — różnice i współdziałanie 80
Tryb ścisły 80
Oczekiwanie na najwyższym poziomie 80
Sposób działania słowa kluczowego this 81
Brakujące odwołania w modułach ES 81
Współdziałanie operacji importowania 82
Importowanie plików JSON 85
Korzystanie z modułów w TypeScripcie 87
Rola kompilatora TypeScriptu 87
Konfiguracja formatu wyjściowego modułu 89
Składnia modułu wejściowego i generowanie danych wyjściowych 89
Rozwiązywanie modułów 90
Podsumowanie 90

ROZDZIAŁ 3

Zdarzenia i funkcje wywołania zwrotnego 91
Wzorzec wywołania zwrotnego 92
Styl przekazywania kontynuacji 93
Synchroniczny czy asynchroniczny? 97
Konwencje funkcji wywołań zwrotnych w Node.js 104
Wzorzec obserwatora 108
Klasa EventEmitter 109
Tworzenie i stosowanie egzemplarza EventEmitter 110
Propagowanie błędów 111
Określenie dowolnego obiektu jako obserwowalnego 111
Ryzyko wycieku pamięci 113
Zdarzenia synchroniczne i asynchroniczne 114
Emiter zdarzeń a funkcje wywołania zwrotnego 116

Łączenie wywołań zwrotnych i zdarzeń ... 117
Podsumowanie ... 121
Ćwiczenia ... 121

ROZDZIAŁ 4
Asynchroniczne wzorce kontroli przepływu z użyciem funkcji wywołań zwrotnych ... 122
Wyzwania związane z programowaniem asynchronicznym ... 123
Tworzenie prostego robota indeksującego strony internetowe ... 123
Piekło wywołań zwrotnych ... 127
Najlepsze praktyki dotyczące wywołań zwrotnych ... 128
Dyscyplina wywołań zwrotnych ... 129
Stosowanie dyscypliny wywołań zwrotnych ... 130
Wzorce przepływu sterowania ... 132
Wykonywanie sekwencyjne ... 132
Wykonywanie współbieżne ... 138
Ograniczone wykonywanie współbieżne ... 144
Podsumowanie ... 152
Ćwiczenia ... 153

ROZDZIAŁ 5
Asynchroniczne wzorce kontroli przepływu z użyciem obietnic i konstrukcji async/await ... 155
Obietnice ... 156
Czym jest obietnica? ... 156
Specyfikacja Promises/A+ i obiekty podobne do obietnic ... 159
API obietnicy ... 160
Tworzenie obietnicy ... 162
Uproszczenie asynchroniczności ... 163
Wykonywanie sekwencyjne i iteracja ... 165
Wykonywanie współbieżne ... 168
Ograniczone wykonywanie współbieżne ... 169
Obietnice leniwe ... 173
Słowa kluczowe async i await ... 177
Funkcje asynchroniczne i wyrażenie await ... 178
Wyrażenie await na najwyższym poziomie ... 179
Obsługa błędów podczas stosowania konstrukcji async/await ... 180
Wykonywanie sekwencyjne i iteracja ... 183
Wykonywanie współbieżne ... 185
Ograniczone wykonywanie współbieżne ... 186

Problem z nieskończonymi łańcuchami rekurencyjnego rozwiązywania obietnic ... 187
Podsumowanie ... 190
Ćwiczenia ... 191

ROZDZIAŁ 6
Programowanie ze strumieniami ... 192

Znaczenie strumieni ... 194
Buforowanie i strumieniowanie ... 194
Efektywność przestrzenna ... 196
Efektywność czasowa ... 197
Komponowalność ... 203
Wprowadzenie do strumieni ... 206
Anatomia strumieni ... 206
Strumienie klasy Readable ... 207
Strumienie klasy Writable ... 214
Strumienie klasy Duplex ... 220
Strumienie klasy Transform ... 221
Strumienie klasy PassThrough ... 228
Strumienie leniwe ... 232
Łączenie strumieni za pomocą potoków ... 233
Asynchroniczne wzorce kontroli przepływu z wykorzystaniem strumieni ... 237
Wykonywanie sekwencyjne ... 237
Wykonywanie współbieżne bez ustalonej kolejności ... 239
Ograniczone wykonywanie współbieżne bez ustalonej kolejności ... 244
Uporządkowane wykonywanie współbieżne ... 245
Wzorce potokowania ... 247
Łączenie strumieni ... 247
Rozwidlanie strumieni ... 251
Scalanie strumieni ... 252
Multipleksowanie i demultipleksowanie ... 254
Narzędzia dla strumieni odczytu ... 260
Mapowanie i przekształcanie ... 260
Filtrowanie i iteracja ... 261
Wyszukiwanie i ocena ... 261
Ograniczanie i redukowanie ... 261
Strumienie internetowe ... 263
Konwersja strumieni Node.js na strumienie internetowe ... 264
Konwersja strumieni internetowych na strumienie Node.js ... 265
Narzędzia dla konsumentów strumieni ... 266
Podsumowanie ... 269
Ćwiczenia ... 269

ROZDZIAŁ 7
Konstrukcyjne wzorce projektowe 271

Fabryka 272
Rozdzielenie tworzenia obiektów od ich implementacji 273
Mechanizm stosowania hermetyzacji 274
Tworzenie prostego profilera kodu 275
W ekosystemie Node.js 278
Budowniczy 278
Implementacja konstruktora obiektów URL 282
W ekosystemie Node.js 285
Konstruktor ujawniający 288
Tworzenie niemodyfikowalnego bufora 289
W ekosystemie Node.js 291
Singleton 292
Łączenie modułów 295
Zależności pojedynczego egzemplarza 296
Wstrzykiwanie zależności 298
Podsumowanie 302
Ćwiczenia 303

ROZDZIAŁ 8
Strukturalne wzorce projektowe 305

Pełnomocnik 305
Metody implementacji pełnomocnika 306
Tworzenie strumienia zapisującego do dziennika 315
Obserwator zmian i Pełnomocnik 317
W ekosystemie Node.js 319
Dekorator 319
Techniki implementacji dekoratorów 320
Dekorowanie bazy danych Level 324
W ekosystemie Node.js 326
Propozycja dekoratorów w specyfikacji ECMAScript 327
Granica między wzorcami projektowymi Pełnomocnik i Dekorator 328
Adapter 329
Korzystanie z biblioteki Level przez API systemu plików 329
W ekosystemie Node.js 332
Podsumowanie 332
Ćwiczenia 333

Kup ksi k Pole ksi k

ROZDZIAŁ 9
Behawioralne wzorce projektowe ... 335
Strategia ... 336
Obiekty konfiguracyjne o wielu formatach ... 338
W ekosystemie Node.js ... 344
Stan ... 344
Implementacja prostego gniazda odpornego na awarie ... 346
W ekosystemie Node.js ... 353
Szablon ... 353
Szablon menedżera konfiguracji ... 354
W ekosystemie Node.js ... 357
Iterator ... 357
Protokół iteratora ... 357
Protokół iterowalny ... 360
Iteratory i obiekty iterowalne jako natywny interfejs JavaScriptu ... 362
Implementacja protokołu iterowalnego dla iteratorów ... 364
Narzędzia do iteracji ... 365
Generatory ... 371
Iteratory asynchroniczne ... 375
Generatory asynchroniczne ... 379
Asynchroniczne iteratory i strumienie Node.js ... 380
Narzędzia do iteracji asynchronicznej ... 381
W ekosystemie Node.js ... 383
Pośrednik ... 383
Oprogramowanie pośredniczące w Expressie ... 384
Wzorzec projektowy Pośrednik ... 385
Tworzenie frameworka oprogramowania pośredniczącego dla ZeroMQ ... 386
W ekosystemie Node.js ... 392
Polecenie ... 393
Wzorzec Zadanie ... 394
Bardziej złożone polecenie ... 395
W ekosystemie Node.js ... 398
Podsumowanie ... 399
Ćwiczenia ... 400

ROZDZIAŁ 10
Testowanie — wzorce i najlepsze praktyki ... 402
Wprowadzenie do testowania oprogramowania ... 403
Definicje ... 404
Rodzaje testów ... 410
Piramida testów ... 415

Kup ksi k Pole ksi k

Tworzenie testów w Node.js ... 417
Nasz pierwszy test jednostkowy ... 417
Program uruchamiający testy w Node.js ... 420
Nasz pierwszy test z użyciem narzędzia testowego Node.js ... 421
Organizacja testów ... 422
Sparametryzowane przypadki testowe ... 426
Wskazówki i podpowiedzi dla testera ... 428
Narzędzie do raportowania testów ... 432
Tworzenie testów jednostkowych ... 436
Testowanie kodu asynchronicznego ... 437
Imitacja ... 442
Tworzenie testów integracyjnych ... 461
Testowanie z wykorzystaniem lokalnej bazy danych ... 461
Testowanie aplikacji internetowej ... 468
Tworzenie testów E2E ... 477
Struktura aplikacji ... 478
Przepływ użytkownika ... 482
Automatyzacja przeglądarki internetowej ... 483
Tworzenie testu E2E przy użyciu Playwrighta ... 485
Podsumowanie ... 497
Ćwiczenia ... 498

ROZDZIAŁ 11
Receptury zaawansowane ... 501
Obsługa komponentów inicjalizowanych asynchronicznie ... 502
Problem z komponentami inicjalizowanymi asynchronicznie ... 503
Lokalna kontrola inicjalizacji ... 505
Opóźnione uruchomienie ... 505
Kolejki przed inicjalizacją ... 507
W ekosystemie Node.js ... 511
Asynchroniczne przetwarzanie wsadowe i buforowanie ... 512
Czym jest asynchroniczne przetwarzanie wsadowe? ... 512
Optymalne buforowanie żądań asynchronicznych ... 514
Serwer API bez pamięci podręcznej i grupowania zapytań ... 516
Przetwarzanie wsadowe i buforowanie z użyciem obietnic ... 519
Anulowanie operacji asynchronicznych ... 522
Podstawowa receptura na tworzenie funkcji, które można anulować ... 523
Opakowywanie wywołań asynchronicznych ... 524
Funkcje asynchroniczne z możliwością anulowania przy użyciu AbortController ... 526

Wykonywanie zadań obciążających procesor 529
Rozwiązywanie problemu sumy podzbioru 529
Przeplatanie z użyciem setImmediate 533
Korzystanie z procesów zewnętrznych 536
Korzystanie z wątków roboczych 543
Uruchamianie w środowisku produkcyjnym zadań obciążających procesor 546
Podsumowanie 546
Ćwiczenia 547

ROZDZIAŁ 12
Skalowalność i wzorce architektoniczne 548

Wprowadzenie do skalowania aplikacji 549
Skalowanie aplikacji utworzonych z użyciem Node.js 549
Trzy wymiary skalowalności 550
Klonowanie i równoważenie obciążenia 552
Moduł cluster 554
Obsługa komunikacji stanowej 563
Skalowanie z użyciem odwrotnego serwera proxy 567
Dynamiczne skalowanie poziome 574
Równoważenie obciążenia między węzłami równorzędnymi 582
Skalowanie aplikacji przy użyciu kontenerów 586
Dekompozycja złożonych aplikacji 594
Architektura monolityczna 595
Architektura mikrousług 596
Wzorce integracji w architekturze mikrousług 600
Podsumowanie 606
Ćwiczenia 606

ROZDZIAŁ 13
Wzorce komunikacji i integracji 608

Podstawy systemu komunikacji 609
Wzorce jednokierunkowe i wzorce żądanie-odpowiedź 609
Typy komunikatów 610
Semantyki dostarczania typu push i pull 612
Kolejki, strumienie i asynchroniczne przekazywanie komunikatów 613
Komunikacja bezpośrednia lub z pośrednikiem 615
Wzorzec Publikowanie-Subskrybowanie 617
Tworzenie minimalistycznej aplikacji czatu działającego w czasie rzeczywistym 618
Wykorzystanie Redisa jako prostego brokera komunikatów 623

Kup ksi k Pole ksi k

Wzorzec pub-sub typu P2P z użyciem ZeroMQ 626
Niezawodne dostarczanie komunikatu za pomocą kolejek 630
Niezawodne przesyłanie komunikatów za pomocą strumieni 640
Wzorce dystrybucji zadań 645
Wzorce fan-out i fan-in w ZeroMQ 647
Potoki i konkurujące konsumenty w protokole AMQP 656
Rozdzielanie zadań za pomocą strumieni Redisa 660
Wzorce żądań i odpowiedzi 666
Identyfikator korelacji 666
Adres zwrotny 671
Podsumowanie 677
Ćwiczenia 677

Skorowidz 679

Rozdział 4

Asynchroniczne wzorce kontroli przepływu z użyciem funkcji wywołań zwrotnych

Prawdziwym wyzwaniem może być przejście z synchronicznego stylu programowania na platformę taką jak Node.js, gdzie standardem są **styl przekazywania kontynuacji** (ang. *continuation-passing style*, CPS) i asynchroniczne API. Kod asynchroniczny utrudnia przewidywanie kolejności wykonywania poleceń. Proste zadania, takie jak iteracja przez zbiór plików, sekwencyjne wykonywanie zadań czy oczekiwanie na zakończenie wielu operacji, wymagają od programistów przyjęcia nowych podejść i technik, aby można było zapobiec tworzeniu nieefektywnego i trudnego w odczycie kodu.

Podczas stosowania funkcji zwrotnych do obsługi asynchronicznej kontroli przepływu najczęstszym błędem jest wpadnięcie w pułapkę „piekła wywołań zwrotnych", gdzie kod rozrasta się poziomo z nadmiernym zagnieżdżeniem oraz utrudnia odczyt i obsługę nawet prostych procedur. Jednak dzięki zastosowaniu odpowiedniego podejścia i wykorzystaniu pewnych wzorców można okiełznać funkcje zwrotne oraz tworzyć czysty i łatwy w zarządzaniu kod asynchroniczny.

W niniejszym rozdziale zagłębimy się w temat funkcji zwrotnych, ich wad i zalet oraz sposobów efektywnego ich wykorzystania. Oto najważniejsze zagadnienia, które omówimy w tym rozdziale:

- wyzwania związane z programowaniem asynchronicznym;
- najlepsze praktyki w zakresie zapobiegania piekłu wywołań zwrotnych i tworzenia wydajnego kodu asynchronicznego;
- popularne wzorce asynchroniczne: wykonanie sekwencyjne (wykonywanie zadań jedno po drugim), iteracja sekwencyjna (przetwarzanie elementów w sekwencji bez równoległego wykonywania zadań), wykonanie współbieżne (uruchamianie wielu zadań jednocześnie), ograniczone wykonanie współbieżne (kontrolowanie liczby jednoczesnych operacji).

Opanowanie tych koncepcji pozwoli Ci tworzyć wydajny i przejrzysty kod asynchroniczny.

Kup ksi k Pole ksi k

Wyzwania związane z programowaniem asynchronicznym

Programowanie asynchroniczne w JavaScripcie może wydawać się proste, ale nie jest pozbawione pułapek. Wykorzystanie domknięć i definicji funkcji anonimowych w miejscu ich użycia pozwala zachować płynne działanie kodu i nie wymaga od programisty przechodzenia do innych fragmentów kodu — mamy tutaj do czynienia z podejściem zgodnym z **regułą KISS** (ang. *Keep It Simple, Stupid*, czyli „zachowaj prostotę, głupcze").

Wskazówka

Reguła KISS to przyjazne przypomnienie, aby priorytetowo traktować prostotę w projektowaniu oprogramowania. Idea ta, ukuta przez inżyniera Kelly'ego Johnsona w firmie Lockheed, zakłada, że rzeczy powinny być łatwe do naprawy, nawet w przypadku ograniczonego dostępu do narzędzi. Dla nas, programistów, oznacza tworzenie przejrzystego i zrozumiałego kodu zamiast skomplikowanych rozwiązań. Dążymy do czytelności, używamy jasnych nazw i trzymamy się sprawdzonych wzorców. Takie podejście nie tylko ogranicza liczbę błędów, ale także ułatwia pracę zespołową i powoduje, że kod jest łatwiejszy w późniejszej obsłudze.

Jednakże tę prostotę często uzyskuje się kosztem modularności, możliwości ponownego użycia kodu i łatwości jego utrzymania. Istnieje ryzyko, że ostatecznie otrzymamy kod, który składa się z wielu zagnieżdżonych wywołań zwrotnych, dużych funkcji i ogólnie jest słabo zorganizowany.

Wprawdzie nie istnieje jedyny „idealny" sposób na problemy związane z kodem asynchronicznym, ale wyposażymy Cię w umiejętności oceny kompromisów i wyboru podejść, które równoważą prostotę z łatwością utrzymania. Wyjaśnimy, jak rozpoznawać potencjalne problemy, zanim się pojawią, przewidywać trudności oraz wdrażać rozwiązania promujące modularność, możliwość ponownego użycia i łatwość utrzymania. Dzięki opanowaniu tych koncepcji będziesz w stanie tworzyć wydajny, czytelny i dobrze ustrukturyzowany kod asynchroniczny, który spełnia wymagania stawiane przed nowoczesnym programowaniem w JavaScripcie — nie poprzez eliminowanie wszystkich kompromisów, ale przez zrozumienie, jak podejmować świadome decyzje o tym, kiedy priorytetowo traktować prostotę, a kiedy złożoność.

Tworzenie prostego robota indeksującego strony internetowe

Aby zaprezentować praktyczny i realistyczny przykład ilustrujący niektóre wyzwania związane z programowaniem asynchronicznym, opracujemy prosty program do pobierania stron internetowych. To będzie działająca w konsoli aplikacja, która przyjmuje adres URL jako dane wejściowe i pobiera jego zawartość lokalnie do jednego lub kilku plików. Takie narzędzie może okazać się przydatne, jeśli chcesz przeglądać stronę internetową w trybie offline lub zachować jej kopię z określonego momentu w czasie.

Kup ksi k Pole ksi k

Kiedyś musiałem zbudować coś bardzo podobnego w pracy! Potrzebowałem programu, który mógłby przeszukać stronę internetową i znaleźć wszystkie niedziałające łącza (błędy 404). Kluczem do obsługi skali witryn, w których ten program był używany — często składających się z setek, a nawet tysięcy stron — okazało się programowanie asynchroniczne.

Tego typu programy są bardzo powszechne w pracy z witrynami internetowymi, a robot indeksujący strony internetowe, którego opracujemy w tym rozdziale, będzie świetnym punktem wyjścia dla Twoich własnych projektów. Możesz łatwo zmienić sposób jego działania lub dodać nowe funkcje, jeśli musisz utworzyć coś podobnego. Na przykład możesz chcieć napisać program, który będzie przeszukiwać wiele stron jednocześnie i wydobywać z nich ustrukturyzowane informacje (czyli tzw. web scraper).

W implementacji będziemy często odwoływać się do lokalnego modułu o nazwie *./utils.js*, zawierającego kilka funkcji pomocniczych, których użyjemy w aplikacji. Dla zachowania zwięzłości projektu pominiemy zawartość tego pliku, ale pełną implementację, wraz z plikiem *package.json* zawierającym kompletną listę zależności, znajdziesz w repozytorium w materiałami dla tej książki (*https://github.com/PacktPublishing/Node.js-Design-Patterns-Fourth-Edition*).

Główna funkcjonalność aplikacji znajduje się w module *spider.js*. Przyjrzyjmy się jego zawartości.

Na początek zaimportujemy niezbędne zależności:

```
import { writeFile } from 'node:fs'
import { dirname } from 'node:path'
import { exists, get, recursiveMkdir, urlToFilename } from './utils.js'
```

Oto krótkie objaśnienie poszczególnych funkcji narzędziowych, które zostały zaimportowane z pliku *./utils.js*:

- `exists` — funkcja oparta na wywołaniu zwrotnym, która sprawdza, czy dany plik istnieje w systemie plików. Przyjmuje ścieżkę dostępu do tego pliku i wykonuje funkcję wywołania zwrotnego wraz z wartością logiczną, która wskazuje, czy dany plik istnieje.
- `get` — funkcja oparta na wywołaniu zwrotnym, która pobiera treść odpowiedzi HTTP dla danego adresu URL. Jako dane wejściowe pobiera adres URL i zwraca bufor reprezentujący treść odpowiedzi.
- `recursiveMkdir` — funkcja oparta na wywołaniu zwrotnym, która rekurencyjnie tworzy wszystkie niezbędne katalogi w określonej ścieżce dostępu.
- `urlToFilename` — funkcja synchroniczna, która przekształca adres URL na poprawną ścieżkę dostępu w systemie plików.

Następnie definiujemy nową funkcję, `spider()`, która przyjmuje dwa parametry: adres URL do pobrania oraz funkcję zwrotną wywoływaną po zakończeniu procesu pobierania:

```
export function spider(url, cb) {
  const filename= urlToFilename(url)
  exists(filename, (err, alreadyExists) => { // 1.
    if (err) { // 1.1.
      cb(err)
```

```
    } else if (alreadyExists) { // 1.2.
      cb(null, filename, false)
    } else { // 1.3.
      console.log(`Pobieranie ${url} do ${filename}`)
      get(url, (err, content) => { // 2.
        if (err) {
          cb(err)
        } else {
          recursiveMkdir(dirname(filename), err => { // 3.
            if (err) {
              cb(err)
            } else {
              writeFile(filename, content, err => { // 4.
                if (err) {
                  cb(err)
                } else {
                  cb(null, filename, true) // 5.
                }
              })
            }
          })
        }
      })
    }
  })
}
```

W tym fragmencie kodu naprawdę dzieje się bardzo wiele, więc szczegółowo omówimy, co zachodzi na każdym etapie:

1. Kod wykorzystuje funkcję `exists()` do sprawdzenia, czy dany adres URL został już wcześniej pobrany. Odbywa się to poprzez weryfikację, czy odpowiadający mu plik jest już obecny na dysku. Na tym etapie możemy otrzymać jeden z trzech możliwych wyników:
   - Wynik 1.1: Otrzymujemy błąd (zmienna `err` jest zdefiniowana). To oznacza, że wystąpił problem podczas uzyskiwania dostępu do systemu plików. W takim przypadku propagujemy błąd z powrotem za pomocą wywołania `cb(err)`.
   - Wynik 1.2: Plik już istnieje na dysku (zmienna `alreadyExists` ma wartość `true`). Nie chcemy pobierać go ponownie. Zatem wystarczy wykonać funkcję wywołania zwrotnego z nazwą pliku i wartością `false` (co informuje komponent wywołujący, że plik nie został teraz pobrany).
   - Wynik 1.3: Plik nie istnieje (zmienna `alreadyExists` ma wartość `false`). W tym przypadku plik musimy pobrać z podanego adresu URL.
2. Do pobrania pliku używamy funkcji `get()`.
3. Musimy upewnić się, że istnieje katalog docelowy. Możemy to zrobić za pomocą funkcji `recursiveMkdir()`.
4. Na koniec jesteśmy gotowi zapisać pobraną zawartość na dysku, co odbywa się przy użyciu funkcji `writeFile()` z Node.js.

Kup ksi k Pole ksi k

5. Jeśli wszystko poszło dobrze, możemy wykonać funkcję wywołania zwrotnego wraz z nazwą pliku i wartością `true` (która tym razem wskazuje, że plik został pobrany).

Aby skorzystać z naszej aplikacji robota indeksującego strony internetowe, możemy opracować prosty interfejs wiersza poleceń (CLI), który jako argument odczytuje adres URL. Można to osiągnąć poprzez utworzenie nowego pliku, *spider-cli.js*:

```
import { spider } from './spider.js'

spider(process.argv[2], (err, filename, downloaded) => {
  if (err) {
    console.error(err)
    process.exit(1)
  }
  if (downloaded) {
    console.log(`Zakończono pobieranie pliku "${filename}"`)
  } else {
    console.log(`Plik "${filename}" został już pobrany`)
  }
})
```

Jeśli skopiowałeś pliki *utils.js* i *package.json* ze wspomnianego wcześniej repozytorium z przykładowymi fragmentami kodu, uruchom polecenie `npm install`, aby pobrać wszystkie niezbędne zależności. Teraz możesz przetestować aplikację robota indeksującego strony internetowe.

Aby uruchomić program, w powłoce przejdź do katalogu projektu i wykonaj następujące polecenie:

```
$ node spider-cli.js https://www.nodejsdesignpatterns.com/
```

To powinno utworzyć plik *www.nodejsdesignpatterns.com/index.html* w bieżącym katalogu roboczym. Następnie możesz sprawdzić zawartość tego pliku i porównać ją z kodem źródłowym w HTML zdalnej strony internetowej oraz poeksperymentować z innymi adresami URL.

Uwaga

Należy pamiętać, że utworzona tutaj prosta aplikacja robota indeksującego strony internetowe wymaga, abyśmy zawsze podawali protokół (na przykład `http://`) w adresie URL. Nie należy też oczekiwać, że łącza HTML zostaną przepisane lub że zasoby takie jak obrazy, filmy, skrypty i arkusze stylów zostaną pobrane. Celowo zachowujemy ten przykład w prostej formie, ponieważ chcemy skoncentrować się na zaprezentowaniu sposobu działania programowania asynchronicznego. Zachęcamy do samodzielnej implementacji funkcjonalności dodatkowej jako ciekawego wyzwania!

Z następnego podrozdziału dowiesz się, jak poprawić czytelność tego kodu źródłowego oraz ogólnie jak kod oparty na wywołaniach zwrotnych utrzymać w postaci maksymalnie czystej i przejrzystej.

Kup książkę Poleć książkę

Piekło wywołań zwrotnych

Przyjrzyj się bliżej funkcji `spider()`, którą zaimplementowaliśmy w poprzednim punkcie. Zauważysz, że nawet w przypadku prostego algorytmu kod staje się głęboko zagnieżdżony i trudny do śledzenia. To częsty problem w przypadku kodu asynchronicznego.

Wyobraź sobie teraz, że moglibyśmy użyć interfejsu blokującego. Kod byłby znacznie prostszy i łatwiejszy do zrozumienia. Aby mieć lepsze wyobrażenie, zobacz, jak mogłaby wyglądać taka implementacja. Przyjmujemy założenie, że mielibyśmy równoważne blokujące API dla wszystkich niezbędnych funkcji pomocniczych:

```
export function spider(url) {
  const filename = urlToFilename(url)
  if (exists(filename)) {
    return false
  } else {
    const content = get(url)
    recursiveMkdir(dirname(filename))
    writeFile(filename, content)
  }
}
```

Zwróć uwagę, że nie musimy nawet jawnie obsługiwać błędów — każdy wyjątek synchroniczny jest automatycznie propagowany w górę stosu wywołań.

Sytuacja wygląda jednak inaczej, gdy mamy do czynienia z programowaniem asynchronicznym, które wykorzystuje styl przekazywania kontynuacji. Definiowanie funkcji zwrotnych w miejscu ich użycia może szybko doprowadzić do nieczytelnego i trudnego w utrzymaniu kodu.

Ten problem, gdzie nadmiar domknięć i zagnieżdżonych wywołań zwrotnych zamienia kod w nieporęczny gąszcz, jest powszechnie znany jako **piekło wywołań zwrotnych** (ang. *callback hell*). To jeden z najbardziej osławionych antywzorców w programowaniu z użyciem Node.js i JavaScriptu. Kod wpadający w tę pułapkę zazwyczaj wygląda mniej więcej tak:

```
asyncFoo(err => {
  asyncBar(err => {
    asyncFooBar(err => {
      //...
    })
  })
})
```

Można zauważyć, że kod utworzony w ten sposób przybiera kształt piramidy (skierowanej w prawo) ze względu na głębokie zagnieżdżenie. Dlatego jest on potocznie nazywany **piramidą zagłady** (ang. *pyramid of doom*).

Najbardziej oczywistym problemem z takim kodem jest jego słaba czytelność. Liczne poziomy zagnieżdżenia mogą znacznie utrudnić rozróżnienie, gdzie kończy się jedna funkcja, a zaczyna druga.

Kolejna kwestia wynika z nakładających się nazw zmiennych w różnych zakresach. Często używamy podobnych, a nawet identycznych nazw dla zmiennych w poszczególnych

Kup ksi k Pole ksi k

zakresach. Typowym przykładem jest argument błędu przekazywany do każdego wywołania zwrotnego. Niektórzy programiści próbują je rozróżniać, stosując podobne warianty, takie jak `err`, `error`, `err1`, `err2` itp. Inni wolą używać tej samej nazwy, np. `err`, we wszystkich zakresach, co powoduje przesłonięcie zmiennej z zakresu zewnętrznego. Żadne z tych podejść nie jest idealne, ponieważ oba zwiększają zamieszanie i ryzyko błędów.

Ponadto domknięcia wiążą się z pewnym narzutem związanym z wydajnością działania i zużyciem pamięci. Mogą również powodować wycieki pamięci, które nie zawsze są łatwe do wykrycia. Każdy kontekst, do którego odwołuje się aktywne domknięcie, jest bowiem zachowywany i nie zostanie usunięty przez mechanizm usuwania nieużytków.

Uwaga

Pamiętaj, że gdy tworzone jest domknięcie, „zamyka" ono otaczające je środowisko leksykalne. To oznacza, że zachowuje dostęp do zmiennych i danych w tym środowisku, nawet po zakończeniu działania funkcji zewnętrznej. Wprawdzie na tym polega istota działania domknięć, ale to wiąże się również z tym, że mechanizm usuwania nieużytków nie może pozbyć się wskazanego kontekstu, dopóki domknięcie jest używane. Zatem w praktyce każda zmienna, obiekt lub inne dane w zasięgu domknięcia pozostaną w pamięci.

Jeśli przyjrzysz się ponownie naszej funkcji `spider()`, zobaczysz, że jest ona doskonałym przykładem piekła wywołań zwrotnych, obrazującym wszystkie problemy, które właśnie omówiliśmy. Teraz zajmiemy się rozwiązaniem tych problemów, z wykorzystaniem wzorców i technik omówionych w dalszej części rozdziału.

Najlepsze praktyki dotyczące wywołań zwrotnych

Teraz, gdy już napotkałeś pierwszy przykład piekła wywołań zwrotnych, wiesz czego unikać. Jednak zarządzanie kodem asynchronicznym niesie ze sobą więcej wyzwań niż tylko zapobieganie głęboko zagnieżdżonym wywołaniom zwrotnym. Istnieje bowiem wiele sytuacji, w których kontrolowanie przepływu wielu zadań asynchronicznych wymaga specyficznych wzorców i technik, szczególnie jeśli pracujesz z czystym JavaScriptem bez użycia zewnętrznych bibliotek.

Rozważmy typowy scenariusz, w którym mamy tablicę adresów URL i chcemy pobrać zawartość każdego z nich **po kolei**, jeden po drugim. Jednym z powodów, dla których moglibyśmy chcieć to zrobić sekwencyjnie, mogą być ograniczenia związane z limitem zapytań (w celu zapobieżenia wysyłaniu więcej niż jednego żądania naraz). Aby rozwiązać ten problem, można by pokusić się o użycie prostej pętli `forEach()`:

```
const urls = ['url1', 'url2', 'url3']
urls.forEach(url => {
  fetch(url, response => {
    console.log(`Pobrano ${url}`)
    // ...Przetwórz odpowiedź
  })
})
```

Kup ksi k Pole ksi k

Jednakże ten kod nie będzie działał zgodnie z oczekiwaniami. Pętla `forEach()` zostanie wykonana synchronicznie i bardzo szybko zainicjuje wszystkie wywołania `fetch`, jedno po drugim. Skoro wywołanie `fetch` jest asynchroniczne, wywołania zwrotne zostaną wykonane w nieokreślonym momencie w przyszłości, bez względu na kolejność elementów w tablicy. To doprowadziłoby do równoczesnego wykonywania żądań i otrzymania wyników, które nie będą uporządkowane, bez wyraźnej kontroli nad wykonaniem kodu.

Aby sekwencyjnie przetwarzać te operacje asynchroniczne, potrzebujemy mechanizmu, który oczekuje na zakończenie każdej operacji przed przejściem do następnej. Popularną techniką rozwiązywania tego typu problemów jest implementacja wzorca podobnego do rekurencji, co omówimy w dalszej części rozdziału.

Teraz natomiast dowiesz się nie tylko, jak uniknąć piekła wywołań zwrotnych, ale także jak zaimplementować niektóre spośród najpopularniejszych wzorców kontroli przepływu — za pomocą jedynie prostego i czystego kodu w JavaScripcie.

Implementacja różnych przepływów sterowania za pomocą wywołań zwrotnych może szybko stać się uciążliwa, jeśli nie zachowamy ostrożności. Jednak zanim zagłębimy się w szczegóły tego, jak używać wywołań zwrotnych do implementacji poszczególnych przepływów sterowania, zróbmy krótką przerwę, aby poznać kilka technik pomocnych w tworzeniu kodu opartego na wywołaniach zwrotnych, który będzie wyższej jakości i łatwiejszy w utrzymaniu. Zastosowanie tych technik ma kluczowe znaczenie, aby praca z wywołaniami zwrotnymi była bardziej przyjemna, a rozwiązanie łatwiejsze w zarządzaniu.

Dyscyplina wywołań zwrotnych

Podczas tworzenia kodu asynchronicznego pierwszą zasadą jest unikanie nadużywania definicji funkcji w miejscu dla wywołań zwrotnych. Wprawdzie to może wydawać się wygodne, ponieważ nie trzeba się martwić o modularność czy możliwość ponownego użycia, ale często powoduje więcej problemów, niż rozwiązuje. W większości przypadków wyeliminowanie piekła wywołań zwrotnych nie wymaga bibliotek, skomplikowanych technik ani zmiany paradygmatu — wystarczy zastosowanie kilku prostych pomysłów.

Oto kilka podstawowych zasad, które mogą zmniejszyć poziom zagnieżdżenia i poprawić ogólną organizację kodu:

- **Wczesne zakończenie działania**. W razie potrzeby używaj polecenia `return`, `continue` lub `break`, aby natychmiast opuścić blok kodu, zamiast zagnieżdżać pełne bloki `if-else`. Dzięki temu kod jest płytszy i łatwiejszy do śledzenia.
- **Użycie nazwanych funkcji dla wywołań zwrotnych**. Wywołania zwrotne przenieś poza domknięcia i przekazuj wyniki pośrednie jako argumenty. Nazwane funkcje zapewniają również bardziej czytelne ślady stosu, co jest pomocne podczas debugowania.
- **Modularność kodu**. Podziel kod na mniejsze funkcje wielokrotnego użytku, gdy tylko to możliwe, aby zwiększyć jego czytelność i łatwość późniejszej obsługi.

Teraz zastosujmy te zasady w praktyce.

Kup ksi k Pole ksi k

Stosowanie dyscypliny wywołań zwrotnych

Aby zilustrować skuteczność wspomnianych zasad dotyczących wywołań zwrotnych, zastosujmy je do wyeliminowania problemu piekła wywołań zwrotnych w naszej aplikacji robota indeksującego strony internetowe.

Przede wszystkim wzorzec sprawdzania błędów możemy zrefaktoryzować poprzez usunięcie polecenia `else`. Takie podejście działa, ponieważ możemy natychmiast zakończyć działanie funkcji po wykryciu w niej błędu. Zatem zamiast pisać taki blok:

```
if (err) {
  cb(err)
} else {
  // Kod do wykonania, gdy nie ma błędów
}
```

kod możemy uprościć do następującej postaci:

```
if (err) {
  return cb(err)
}
// Kod do wykonania, gdy nie ma błędów
```

Takie rozwiązanie często określa się jako **zasadę wczesnego zwrotu**. Dzięki temu prostemu zabiegowi od razu redukujemy poziom zagnieżdżenia funkcji. Jest to łatwe i nie wymaga skomplikowanej refaktoryzacji.

Uwaga

W przypadku opisanej optymalizacji często popełnianym błędem jest zapomnienie o zakończeniu działania funkcji po wykonaniu wywołania zwrotnego. Podczas obsługi błędów poniższy fragment kodu jest typowym źródłem problemów:

```
if (err) {
  cb(err)
}
// Kod do wykonania, gdy nie ma błędów
```

Nie powinniśmy zapominać, że wykonywanie funkcji będzie kontynuowane nawet po uruchomieniu funkcji wywołania zwrotnego. Dlatego tak ważne jest, aby wstawić polecenie `return`, które uniemożliwi wykonywanie pozostałego fragmentu kodu funkcji. Warto też zauważyć, że wartość zwracana przez funkcję nie ma większego znaczenia — rzeczywisty wynik (lub błąd) jest generowany asynchronicznie i przekazywany do funkcji wywołania zwrotnego. Wartość zwrotna funkcji asynchronicznej jest zazwyczaj ignorowana. Ta właściwość pozwala stosować skróty, takie jak następujący:

```
return cb(...)
```

W przeciwnym razie musielibyśmy utworzyć nieco bardziej rozwlekły kod, na przykład:

```
cb(...)
return
```

Kup ksi k Pole ksi k

W ramach drugiej optymalizacji funkcji `spider()` możemy zidentyfikować fragmenty kodu, które nadają się do wielokrotnego użycia. Na przykład funkcjonalność zapisywania danego ciągu tekstowego do pliku (przy czym należy się upewnić, że wskazany katalog zostanie utworzony, jeśli to konieczne) można łatwo wyodrębnić do oddzielnej funkcji:

```
function saveFile(filename, content, cb) {
  recursiveMkdir(dirname(filename), err => {
    if (err) {
      return cb(err)
    }
    writeFile(filename, content, cb)
  })
}
```

Zgodnie z tą samą zasadą możemy zdefiniować ogólną funkcję `download()`, która jako parametry wejściowe pobiera adres URL i nazwę pliku, a następnie pobiera zawartość z podanego adresu i zapisuje ją do wskazanego pliku. Wewnętrznie możemy wykorzystać funkcję `saveFile()`, którą wcześniej utworzyliśmy:

```
function download(url, filename, cb) {
  console.log(`Pobieranie ${url} do ${filename}`)
  get(url, (err, content) => {
    if (err) {
      return cb(err)
    }
    saveFile(filename, content, err => {
      if (err) {
        return cb(err)
      }
      cb(null, content)
    })
  })
}
```

Teraz kod jest bardziej modularny i przejrzysty. Aby zakończyć refaktoryzację, zmodyfikujemy funkcję `spider()`:

```
export function spider(url, cb) {
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    if (err) {
      return cb(err)
    }
    if (alreadyExists) {
      return cb(null, filename, false)
    }
    download(url, filename, err => {
      if (err) {
        return cb(err)
      }
      cb(null, filename, true)
    })
  })
}
```

Kup ksi k Pole ksi k

Funkcjonalność i interfejs funkcji `spider()` pozostają niezmienione. Natomiast organizacja kodu została poprawiona. Dzięki zastosowaniu zasady wczesnego zwracania wyniku oraz innych technik porządkowania wywołań zwrotnych znacząco zmniejszyliśmy poziom zagnieżdżenia kodu, a jednocześnie zwiększyliśmy jego możliwości w zakresie ponownego użycia i testowania. Moglibyśmy rozważyć wyeksportowanie funkcji `saveFile()` i `download()`, a następnie przeniesienie ich do wspólnej biblioteki narzędziowej, aby tym samym umożliwić ich wykorzystanie w różnych modułach. To pozwoliłoby również testować te funkcje w izolacji, co przyczyniłoby się do poprawy jakości kodu.

Przedstawiona refaktoryzacja pokazuje, że często wystarczy odrobina dyscypliny, aby uniknąć nadmiernego stosowania domknięć i funkcji anonimowych. To proste i skuteczne podejście, które wymaga minimalnego wysiłku i nie opiera się na bibliotekach zewnętrznych.

Teraz, gdy już wiesz, jak tworzyć czysty i asynchroniczny kod z wykorzystaniem wywołań zwrotnych, możemy przyjrzeć się popularnym wzorcom asynchronicznym, takim jak wykonywanie sekwencyjne i współbieżne.

Wzorce przepływu sterowania

W tym podrozdziale przyjrzymy się asynchronicznym wzorcom kontroli przepływu. Rozpoczniemy od analizy przepływu wykonywania sekwencyjnego.

Wykonywanie sekwencyjne

Sekwencyjne wykonywanie zestawu zadań oznacza uruchamianie ich po kolei, jedno za drugim. Kolejność wykonywania ma znaczenie i musi być zachowana, ponieważ wynik jednego zadania może wpływać na sposób wykonania następnego. W sposób graficzny ta koncepcja została zilustrowana na rysunku 4.1.

Rysunek 4.1. Przykład przepływu wykonywania sekwencyjnego z trzema zadaniami

Istnieją różne warianty tego przepływu:

- Wykonywanie zestawu znanych zadań w sekwencji, bez przekazywania danych między nimi.
- Wykorzystanie wyniku jednego zadania jako danych wejściowych dla następnego (jest to znane jako łańcuch, potok lub kaskada).
- Iterowanie po kolekcji przy jednoczesnym wykonywaniu asynchronicznego zadania na każdym elemencie, jeden po drugim.

Kup książkę Poleć książkę

Wykonywanie sekwencyjne jest zwykle główną przyczyną problemu piekła wywołań zwrotnych podczas korzystania z asynchronicznego programowania opartego stylu przekazywania kontynuacji (CPS).

Wykonywanie znanego zestawu zadań w określonej kolejności

Być może nie zdawałeś sobie z tego sprawy, ale w poprzednim punkcie przedstawiliśmy już koncepcję przepływu wykonania sekwencyjnego podczas implementacji funkcji `spider()`. Jeśli się nad tym zastanowisz, zauważysz, że robot indeksujący strony internetowe wykonuje kilka zadań asynchronicznych w określonej kolejności, przy czym każde zadanie kończy się przed rozpoczęciem następnego: sprawdzanie, czy plik już istnieje, pobieranie treści z zewnętrznego adresu URL i zapisywanie tej treści do pliku. Przestrzegając kilku prostych zasad, mogliśmy zorganizować te zadania jako przepływ sekwencyjny. Teraz, korzystając z tego kodu jako punktu odniesienia, możemy uogólnić nasze rozwiązanie według następującego wzorca:

```
function task1(cb) {
  asyncOperation(() => {
    task2(cb) // Wywołanie zadania task2 z bieżącym wywołaniem zwrotnym
  })
}

function task2(cb) {
  asyncOperation(() => {
    task3(cb) // Wywołanie zadania task3 z bieżącym wywołaniem zwrotnym
  })
}

function task3(cb) {
  asyncOperation(() => {
    cb() // Kończy wykonanie zadań i uruchamia wywołanie zwrotne
  })
}

task1(() => {
  // Wykonywane po zakończeniu wykonywania zadań 1., 2. i 3.
  console.log('Zadania 1., 2. i 3. zostały wykonane')
})
```

Powyższy wzorzec pokazuje, jak każde zadanie uruchamia kolejne po zakończeniu ogólnej operacji asynchronicznej. To podkreśla modularyzację zadań i wskazuje, że domknięcia nie zawsze są konieczne do obsługi kodu asynchronicznego.

Iteracja sekwencyjna

Wzorzec wykonywania sekwencyjnego sprawdza się świetnie, kiedy z góry wiemy, jakie zadania mają zostać wykonane i ile ich jest, zwłaszcza gdy liczba zadań jest stosunkowo niewielka. To pozwala bezpośrednio zakodować wywołania każdego kolejnego zadania w sekwencji. Ale co się dzieje, kiedy chcemy wykonać operację asynchroniczną dla każdego elementu w kolekcji? W takich sytuacjach nie możemy już zakodować sekwencji zadań na stałe, lecz musimy zbudować ją dynamicznie.

Kup ksi k Pole ksi k

Wersja druga robota indeksującego strony internetowe

Aby zilustrować iterację sekwencyjną, wprowadźmy nową funkcjonalność do naszej aplikacji robota indeksującego strony internetowe: rekurencyjne pobieranie wszystkich łączy na stronie internetowej, o ile pozostają one w obrębie tej samej domeny. W końcu to robot indeksujący — potrafi przemierzać sieć i zagłębiać się w nią, podążając za łączami! Aby to osiągnąć, wyodrębnimy wszystkie łącza z bieżącej strony i spowodujemy, że robot będzie podążał za każdym z nich i kolejno je pobierał w sposób rekurencyjny.

Pierwszym krokiem jest modyfikacja funkcji `spider()`, aby uruchamiała rekurencyjne pobieranie łączy ze strony przy użyciu nowej funkcji, `spiderLinks()`, którą wkrótce zdefiniujemy.

Kluczową decyzją projektową jest zagwarantowanie, że robot indeksujący strony internetowe nie utknie w pętli działającej w nieskończoność. Aby temu zapobiec, wprowadzimy parametr `maxDepth`, który ograniczy głębokość rekurencji. Oto zaktualizowany kod funkcji `spider()`:

```
export function spider(url, maxDepth, cb) {
  const filename= urlToFilename(url)

  exists(filename, (err, alreadyExists) => {
    if (err) {
      // Błąd podczas sprawdzania pliku
      return cb(err)
    }

    if (alreadyExists) {
      if (!filename.endsWith('.html')) {
        // Pomijanie zasobów innych niż HTML
        return cb()
      }

      // Jeśli strona została już pobrana, odczytaj jej zawartość i pobierz łącza
      return readFile(filename, 'utf8', (err, fileContent) => {
        if (err) {
          // Błąd podczas odczytu pliku
          return cb(err)
        }
        return spiderLinks(url, fileContent, maxDepth, cb)
      })
    }

    // Plik nie istnieje, pobierz go
    download(url, filename, (err, fileContent) => {
      if (err) {
        // Błąd pobierania pliku
        return cb(err)
      }
      // W przypadku pliku HTML należy go przeanalizować
      if (filename.endsWith('.html')) {
        return spiderLinks(url, fileContent.toString('utf8'),
          maxDepth, cb)
      }
```

Kup ksi k Pole ksi k

```
        // W przeciwnym razie zakończ działanie
        return cb()
      })
    })
  }
```

W całym kodzie dodaliśmy komentarze, dzięki którym łatwiej będzie zrozumieć sposób działania nowej wersji funkcji.

W następnym podpunkcie wyjaśnimy, jak została zaimplementowana funkcja `spider` ↪`Links()`.

Sekwencyjne przeglądanie łączy

Teraz możemy zbudować rdzeń nowej wersji robota indeksującego strony internetowe: funkcję `spiderLinks()`. Pobiera ona wszystkie łącza ze strony HTML przy użyciu sekwencyjnego algorytmu asynchronicznej iteracji. Przyjrzyjmy się uważnie, jak ją zdefiniować:

```
function spiderLinks(currentUrl, body, maxDepth, cb) {
  if (maxDepth === 0) { // 1.
    return process.nextTick(cb)
  }

  const links = getPageLinks(currentUrl, body) // 2.
  if (links.length === 0) {
    return process.nextTick(cb)
  }

  function iterate(index) { // 3.
    if (index === links.length) {
      return cb()
    }

    spider(links[index], maxDepth - 1, err => { // 4.
      if (err) {
        return cb(err)
      }
      iterate(index + 1)
    })
  }

  iterate(0) // 5.
}
```

Oto kluczowe kroki, które należy zrozumieć w przypadku tej nowej funkcji:

1. Przypadek bazowy. Jeśli zmienna `maxDepth` osiągnie wartość `0`, kończymy przeszukiwanie. Zwróć uwagę, że funkcję zwrotną wywołujemy asynchronicznie, aby uniknąć problemu Zalgo.
2. Wszystkie łącza ze strony internetowej wyodrębniamy za pomocą funkcji `getPageLinks()`, która jest synchroniczna i zwraca tylko te łącza, które wskazują tę samą nazwę hosta (ta funkcja jest zaimplementowana w pliku *utils.js*). Jeśli nie ma żadnych łączy, kończymy działanie procesu.

Kup ksi k Pole ksi k

3. Używamy lokalnej funkcji o nazwie `iterate()` do przeglądania łączy. Przyjmuje ona indeks następnego łącza do przetworzenia. Funkcja najpierw sprawdza, czy indeks jest równy długości tablicy łączy. Jeśli tak, oznacza to, że przetworzyliśmy wszystkie elementy i możemy zakończyć działanie za pomocą wywołania `cb()`.
4. Na tym etapie wszystko jest gotowe do przetworzenia łącza. Wywołujemy funkcję `spider()`, zmniejszamy maksymalną głębokość rekurencji (`maxDepth - 1`) i kontynuujemy następny krok iteracji, gdy operacja się zakończy.
5. Ostatnim krokiem w trakcie działania funkcji `spiderLinks()` jest rozpoczęcie iteracji za pomocą wywołania `iterate(0)`.

Przedstawiony tutaj algorytm pozwala iterować po tablicy poprzez sekwencyjne wykonywanie operacji asynchronicznej, którą w naszym przypadku jest funkcja `spider()`.

Teraz zaktualizujmy plik *spider-cli.js*, aby umożliwić określenie poziomu zagnieżdżenia jako dodatkowego argumentu interfejsu wiersza poleceń (CLI):

```
import { spider } from './spider.js'

const url = process.argv[2]
const maxDepth = Number.parseInt(process.argv[3], 10) || 1

spider(url, maxDepth, err => {
  if (err) {
    console.error(err)
    process.exit(1)
  }

  console.log('Zakończono pobieranie')
})
```

Teraz możemy przetestować nową wersję aplikacji robota indeksującego strony internetowe i zaobserwować, jak pobiera ona wszystkie łącza ze strony internetowej w sposób rekurencyjny, jeden po drugim. Aby zatrzymać proces — który w przypadku wielu łączy może trwać dłuższy czas — możesz nacisnąć klawisze *Ctrl+C*. Jeśli chcesz później wznowić pracę, możesz ponownie uruchomić robota i podać ten sam adres URL, którego użyłeś podczas poprzedniego uruchomienia programu.

Uwaga

Teraz, gdy nasz robot indeksujący strony internetowe ma możliwość pobrania całej witryny, należy korzystać z niego ostrożnie. Na przykład unikaj ustawiania wysokiego poziomu maksymalnej głębokości lub pozostawiania robota uruchomionego zbyt długo. Przeciążanie serwera tysiącami żądań jest nie tylko nieuprzejme, ale w niektórych przypadkach może być nawet nielegalne i zostać uznane za atak typu DoS (odmowa usługi). Korzystaj z robota odpowiedzialnie!

Wzorzec

Kod funkcji `spiderLinks()` z poprzedniego podpunktu jest dobrym przykładem tego, jak można iterować po kolekcji, wykonując jednocześnie operację asynchroniczną. Zwróć uwagę, że to jest wzorzec możliwy do dostosowania w każdej innej sytuacji, w której musimy asynchronicznie przeglądać elementy kolekcji lub ogólnie listę zadań. Ten wzorzec można uogólnić w następujący sposób:

Kup ksi k Pole ksi k

```
function iterate (index) {
  if (index === tasks.length) {
    return finish()
  }
  const task = tasks[index]
  task(() => iterate(index + 1))
}

function finish () {
  // Iteracja zakończona
}

iterate(0)
```

Uwaga

Warto zauważyć, że tego typu algorytmy stają się rekurencyjne, jeśli operacja `task()` jest synchroniczna. W takim przypadku stos nie będzie się zwijał po każdym cyklu, co może skutkować ryzykiem osiągnięcia maksymalnej wielkości stosu wywołań.

Przedstawiony wzorzec można rozszerzyć lub dostosować do wielu powszechnych potrzeb. Oto kilka przykładów:

- Asynchroniczne mapowanie wartości tablicy.
- Przekazywanie wyników jednej operacji do następnej podczas iteracji w celu zaimplementowania asynchronicznej wersji algorytmu redukcji.
- Wcześniejsze zakończenie pętli, jeśli spełniony zostanie określony warunek (asynchroniczna implementacja metody `Array.some()`).
- Iteracja po nieskończonej liczbie elementów.

Możemy także bardziej uogólnić rozwiązanie — wystarczy je opakować w funkcję o sygnaturze podobnej do poniższej:

```
iterateSeries(kolekcja, wywołanie_zwrotne_iteratora, ostateczne_wywołanie_zwrotne)
```

W tym przypadku `kolekcja` to faktyczny zbiór danych, po którym chcemy iterować, `wywołanie_zwrotne_iteratora` to funkcja wykonywana dla każdego elementu, a `ostateczne_wywołanie_zwrotne` to funkcja wykonywana po przetworzeniu wszystkich elementów lub w przypadku wystąpienia błędu. Implementację tej funkcji pomocniczej pozostawiamy jako ćwiczenie dla czytelnika.

Uwaga

Wzorzec iteratora sekwencyjnego

Umożliwia wykonywanie zadań w kolekcji, jedno po drugim, i zapewniaj kontrolowany przebieg operacji. Jest to realizowane za pomocą funkcji iteratora, która automatycznie uruchamia następne zadanie w sekwencji, gdy tylko bieżące zostanie zakończone.

W następnym punkcie omówimy wzorzec wykonania współbieżnego, który okazuje się wygodniejszy, gdy kolejność wykonywania poszczególnych zadań nie ma znaczenia.

Kup książkę Poleć książkę

Wykonywanie współbieżne

Istnieją sytuacje, w których kolejność wykonywania zestawu zadań asynchronicznych nie ma znaczenia i nie zachodzą między nimi logiczne powiązania ani zależności danych. Chcemy jedynie otrzymać powiadomienie, gdy wszystkie uruchomione zadania zostaną zakończone. Takie sytuacje lepiej obsługiwać za pomocą przepływu wykonania współbieżnego.

Na tym etapie książki kilkakrotnie użyliśmy słów **równoległy** i **współbieżny**. Wprawdzie ich znaczenie może się wydawać podobne, ale różnica między nimi jest ważną koncepcją do zrozumienia, szczególnie w kontekście Node.js.

Jak już to wyjaśniliśmy w rozdziale 1., platforma Node.js jest jednowątkowa i dlatego zrozumienie koncepcji współbieżności jest szczególnie istotne. Zanim przejdziemy dalej, spróbujmy zdefiniować te dwa pojęcia za pomocą analogii.

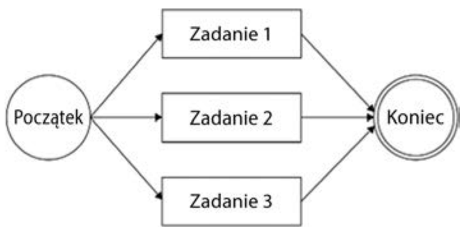
Wyobraź sobie, że prowadzisz restaurację i otrzymujesz dwa zamówienia jednocześnie. W kuchni są dwa sposoby, aby sobie z tym poradzić:

- **Równoległość**. Jeśli masz dwóch kucharzy, każdy może przygotowywać zamówienie jednocześnie, pracując całkowicie niezależnie. To przypomina posiadanie wielu rdzeni procesora, które równolegle wykonują zadania.
- **Współbieżność**. Jeśli jest tylko jeden kucharz, musi on efektywnie podzielić swój czas. Podczas oczekiwania na zagotowanie wody do jednego dania może zacząć krojenie składników do drugiego. Kucharz nie pracuje nad obydwoma zadaniami jednocześnie, ale robi postępy w obu. Tak działa pętla zdarzeń, która efektywnie przełącza się między zadaniami podczas oczekiwania na zakończenie wolniejszego, takiego jak operacja wejścia-wyjścia.

Platforma Node.js jest zbudowana na JavaScripcie, który działa z użyciem jednowątkowej pętli zdarzeń. Zatem domyślnie Node.js doskonale radzi sobie z wykonywaniem współbieżnym i obsługuje wiele zadań bez blokowania wątku głównego.

Chociaż równoległość można osiągnąć za pomocą wątków roboczych lub poprzez uruchomienie wiele procesów, współbieżność jest często bardziej wydajna i lżejsza w wielu rzeczywistych zastosowaniach, takich jak obsługa wielu żądań sieciowych lub zapytań do bazy danych.

Równoległe wykonywanie wielu zadań zostało graficznie pokazane na rysunku 4.2.

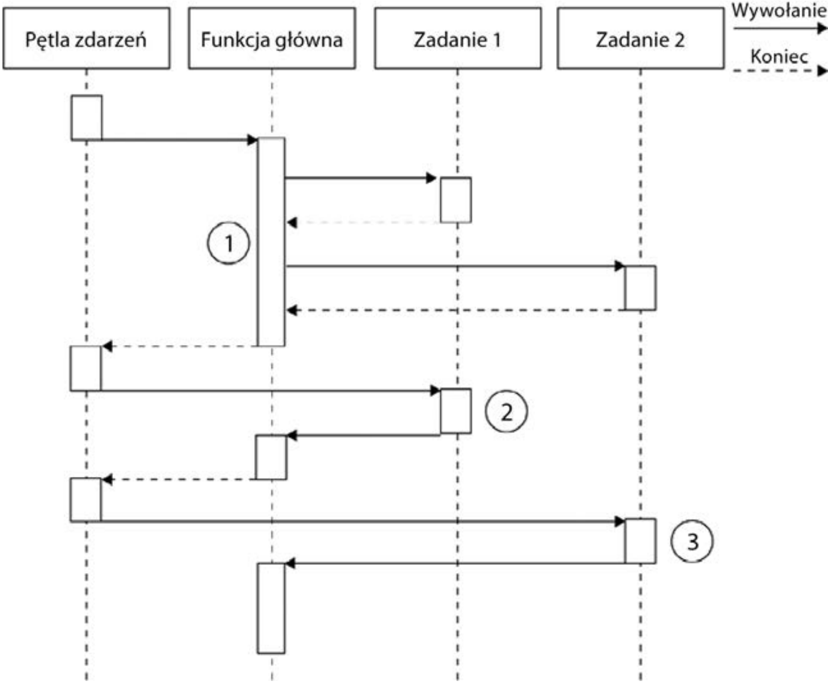


Rysunek 4.2. Przykład wykonywania równoległego z trzema zadaniami

Kup ksi k Pole ksi k

Chociaż ten diagram jest zazwyczaj używany do przedstawienia równoległego wykonywania wielu zadań, może również pomóc w opisaniu współbieżności na wysokim poziomie, gdzie wiele zadań aktywnie postępuje w ramach pętli zdarzeń. Kluczowa różnica między wykonywaniem równoległym i współbieżnym leży w sposobie, w jaki dokonuje się postęp: wykonywanie równoległe uruchamia zadania jednocześnie, natomiast współbieżność polega na efektywnym przełączaniu się między zadaniami, by utrzymać ich ciągły postęp.

Aby lepiej zrozumieć tę koncepcję, przyjrzyjmy się innemu diagramowi, który pokazuje, jak dwa zadania asynchroniczne mogą działać współbieżnie w programie utworzonym z użyciem Node.js (rysunek 4.3).



Rysunek 4.3. Przykład współbieżnego wykonywania zadań asynchronicznych

W sytuacji pokazanej na rysunku 4.3 przedstawiono funkcję **Main**, która asynchroniczne uruchamia dwa zadania:

1. Funkcja **Main** inicjuje wykonanie **Zadania 1.** i **Zadania 2.** Ponieważ uruchamiają one operacje asynchroniczne, natychmiast zwracają kontrolę do funkcji **Main**, która następnie przekazuje ją do pętli zdarzeń.
2. Gdy operacja asynchroniczna **Zadania 1.** zostaje zakończona, pętla zdarzeń przekazuje mu kontrolę. Po ukończeniu wewnętrznego przetwarzania synchronicznego,**Zadanie 1.** powiadamia funkcję **Main**.
3. Gdy operacja asynchroniczna zainicjowana przez **Zadanie 2.** zostaje zakończona, pętla zdarzeń wykonuje jej funkcję wywołania zwrotnego i przekazuje kontrolę z powrotem do **Zadania 2.** Po zakończeniu **Zadania 2.** funkcja **Main** jest

ponownie powiadamiana. W tym momencie funkcja **Main** wie, że zakończyło się wykonywanie obu zadań, więc może kontynuować działanie bądź przekazać wyniki operacji do innej funkcji wywołania zwrotnego.

Podsumowując: na platformie Node.js operacje asynchroniczne zazwyczaj wykonujemy współbieżnie, ponieważ ich współbieżność jest obsługiwana wewnętrznie przez nieblokujące API. Natomiast operacje synchroniczne (blokujące) nie mogą być w łatwy sposób wykonywane równolegle bądź współbieżnie, o ile nie będą przeplatane operacją asynchroniczną albo funkcjami typu `setTimeout()` lub `setImmediate()`. Te techniki omawiamy szczegółowo w rozdziale 11.

Wersja trzecia robota indeksującego strony internetowe

Nasza aplikacja robota indeksującego strony internetowe wydaje się idealnym kandydatem do zastosowania koncepcji wykonywania równoległego. Dotychczas ta przykładowa aplikacja rekurencyjnie pobierała powiązane strony, co odbywało się w sposób sekwencyjny. Możemy łatwo poprawić wydajność tego procesu — wystarczy jednocześnie pobierać wszystkie powiązane strony.

Aby to zrobić, należy zmodyfikować funkcję `spiderLinks()`, aby jednocześnie uruchamiała wszystkie zadania `spider()`, a końcową funkcję wywołania zwrotnego wykonywała dopiero po zakończeniu wszystkich zadań. Zmodyfikujmy więc funkcję `spiderLinks()` w następujący sposób:

```
function spiderLinks(currentUrl, body, maxDepth, cb) {
  if (maxDepth === 0) {
    return process.nextTick(cb)
  }

  const links = getPageLinks(currentUrl, body)
  if (links.length === 0) {
    return process.nextTick(cb)
  }

  let completed = 0
  let hasErrors = false // 3.

  function done(err) { // 2.
    if (err) {
      hasErrors = true
      return cb(err)
    }
    if (++completed === links.length && !hasErrors) {
      return cb()
    }
  }

  for (const link of links) { // 1.
    spider(link, maxDepth - 1, done)
  }
}
```

Omówmy wprowadzone zmiany:

1. Jak już wcześniej wspomnieliśmy, zadania `spider()` są teraz uruchamiane jednocześnie. Jest to możliwe dzięki przeprowadzeniu zwykłej iteracji przez tablicę łączy i uruchamianiu poszczególnych zadań bez oczekiwania na zakończenie poprzedniego.
2. Następnie, aby aplikacja poczekała na zakończenie wszystkich zadań, funkcji `spider()` przekazujemy specjalną funkcję wywołania zwrotnego o nazwie `done()`. Zwiększa ona wartość licznika po zakończeniu każdego zadania. Gdy liczba ukończonych pobrań osiągnie wielkość tablicy łączy, następuje wykonanie końcowej funkcji wywołania zwrotnego, `cb()`.
3. Zmienna `hasErrors` jest potrzebna, ponieważ jeśli którekolwiek z równoległych zadań zakończy się niepowodzeniem, chcemy natychmiast wykonać funkcję wywołania zwrotnego wraz z danym błędem. Ponadto musimy się upewnić, że pozostałe zadania, które mogą wciąż działać, nie wykonają ponownie danej funkcji wywołania zwrotnego.

Jeżeli teraz po wprowadzeniu tych zmian ponownie uruchomimy robota indeksującego strony internetowe, zauważymy ogromną poprawę szybkości całego procesu, ponieważ wszystkie operacje pobierania będą wykonywane równolegle, bez oczekiwania na przetworzenie poprzedniego łącza.

Wzorzec

Na koniec możemy wyodrębnić nasz elegancki wzorzec dla przepływu wykonania współbieżnego. Ogólną wersję tego wzorca przedstawmy za pomocą poniższego fragmentu kodu:

```
const tasks = [ /* ... */ ]

let completed = 0
for (const task of tasks) {
  task(() => {
    if (++completed === tasks.length) {
      finish()
    }
  })
}

function finish () {
  // Wszystkie zadania zostały ukończone
}
```

Dzięki wprowadzeniu niewielkich modyfikacji ten wzorzec możemy dostosować do gromadzenia wyników poszczególnych zadań w kolekcji, filtrowania lub mapowania elementów tablicy bądź wykonywania funkcji wywołania zwrotnego `finish()` natychmiast po zakończeniu jednego lub określonej liczby zadań (ta ostatnia sytuacja jest nazywana **wyścigiem konkurencyjnym**).

Uwaga

Wzorzec nieograniczonego wykonywania współbieżnego

Ten wzorzec polega na równoczesnym uruchamianiu zestawu zadań asynchronicznych poprzez natychmiastowe rozpoczęcie ich wszystkich i oczekiwanie na ich zakończenie. Wszystkie zadania są uruchamiane od razu, a ich wykonywanie jest śledzone poprzez zliczanie, ile razy wykonywane są ich funkcje wywołania zwrotnego.

Gdy mamy wiele zadań wykonywanych jednocześnie, mogą wystąpić sytuacje wyścigu, czyli rywalizacja o dostęp do zasobów zewnętrznych (np. plików lub rekordów w bazie danych). W następnym podpunkcie omówimy sytuacje wyścigu na platformie Node.js oraz przyjrzymy się wybranym technikom ich identyfikowania i rozwiązywania.

Rozwiązywanie problemów z warunkami wyścigu w zadaniach współbieżnych

Uruchamianie zestawu zadań równolegle może powodować problemy w przypadku korzystania z blokujących operacji wejścia-wyjścia w połączeniu z wieloma wątkami. Jednak, jak przed chwilą pokazaliśmy, na platformie Node.js sytuacja wygląda zupełnie inaczej. Jednoczesne wykonywanie wielu zadań asynchronicznych jest w rzeczywistości proste i oszczędne pod względem zasobów.

To jedna z najważniejszych zalet środowiska Node.js, ponieważ powoduje, że równoległe wykonywanie wielu zadań staje się powszechną praktyką, a nie skomplikowaną techniką stosowaną tylko w razie konieczności.

Inną istotną cechą modelu współbieżności Node.js jest sposób, w jaki platforma radzi sobie z synchronizacją zadań i warunkami wyścigu. W środowisku wielowątkowym zarządzanie zasobami współdzielonymi zazwyczaj wymaga mechanizmów synchronizacji, takich jak blokady, muteksy, semafory i monitory. Wprawdzie pomagają one koordynować dostęp do współdzielonych danych, ale mogą wprowadzać znaczną złożoność i obciążenie, które prowadzi do obniżenia wydajności działania.

Wracając do naszej analogii kuchennej, wyobraźmy sobie dwóch kucharzy pracujących równolegle, z których każdy przygotowuje własne danie. Jeśli obaj w tym samym czasie muszą skorzystać z tego samego zlewu, by umyć składniki, to nie będą mogli jednocześnie kontynuować pracy — muszą albo się zmieniać, albo zaryzykować, że będą sobie nawzajem przeszkadzać. W programowaniu wielowątkowym mechanizmy takie jak blokady służą do zarządzania taką rywalizacją i zapewniają, że w danym momencie tylko jedno zadanie uzyska dostęp do zasobu współdzielonego. Jednak źle zarządzana synchronizacja może prowadzić do nieefektywności — jakby jeden z kucharzy blokował zlew zbyt długo, a tym samym spowalniał pracę całej kuchni.

Na platformie Node.js zazwyczaj nie potrzebujemy skomplikowanych mechanizmów synchronizacji, ponieważ wszystko działa w ramach pojedynczego wątku. To jednak nie oznacza, że nie będziemy mieć do czynienia z sytuacjami wyścigu — wręcz przeciwnie, mogą one być dość powszechne. Źródłem problemu jest opóźnienie między wywołaniem operacji asynchronicznej i powiadomieniem o jej wyniku.

Kup ksi k Pole ksi k

Wskazówka

Sytuacja wyścigu to stan, w którym zachowanie programu zależy od czasu wykonania równoległych operacji uzyskujących dostęp do zasobów współdzielonych.

Aby zobaczyć konkretny przykład, ponownie odwołamy się do naszej aplikacji robota indeksującego strony internetowe, w szczególności do jej ostatniej wersji, która zawiera sytuację wyścigu. Czy potrafisz ją dostrzec? Poświęć kilka minut na zastanowienie się i spróbuj odgadnąć, na czym polega problem. Poczekamy tutaj na Ciebie!

Problem, o którym mówimy, tkwi w funkcji `spider()`, gdzie sprawdzamy, czy plik już istnieje, zanim zaczniemy pobierać odpowiedni adres URL:

```
export function spider(url, maxDepth, cb) {
  const filename= urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    //...
    if (alreadyExists) {
      //...
    } else {
      download(url, filename, (err, fileContent) => {
        //...
```

Problem polega na tym, że dwa zadania robota indeksującego strony internetowe, które przetwarzają ten sam adres URL, mogą wywołać metodę `exists()` dla tego samego pliku, zanim jedno z zadań ukończy pobieranie i utworzy plik, co z kolei spowoduje, że oba zadania rozpoczną pobieranie. Tę sytuację ilustruje rysunek 4.4.

Rysunek 4.4. Przykład stanu wyścigu w funkcji spider()

Sytuacja przedstawiona na rysunku 4.4 pokazuje, jak **Zadanie 1.** i **Zadanie 2.** są przeplatane w pojedynczym wątku Node.js, a także jak operacja asynchroniczna może wprowadzić stan wyścigu. W omawianym przypadku dwa zadania robota kończą się pobraniem tego samego pliku.

Jak możemy to wyeliminować? Odpowiedź jest prostsza, niż mogłoby się wydawać. Otóż potrzebujemy tylko zmiennej, która wzajemnie wykluczy wiele zadań `spider()` uruchomionych dla tego samego adresu URL. Można to zrealizować za pomocą następującego fragmentu kodu:

```
const spidering = new Set()
function spider (url, nesting, cb) {
  if (spidering.has(url)) {
    return process.nextTick(cb)
```

```
}
spidering.add(url)
//...
```

Jeśli adres URL znajduje się już w globalnym zbiorze przeszukiwanych stron, `spidering`, funkcja natychmiast kończy działanie. W przeciwnym razie dodajemy adres URL do zbioru i przechodzimy do pobierania pliku. Ponieważ nie chcemy pobierać tego samego adresu URL więcej niż raz, nie ma potrzeby usuwania adresów ze zbioru.

Wskazówka

Jeśli tworzysz robota, który będzie musiał pobrać setki tysięcy stron internetowych, usuwanie pobranych adresów URL ze zbioru po pobraniu pliku pomoże utrzymać liczność zbioru, a tym samym zużycie pamięci, na stałym poziomie bez nieograniczonego wzrostu.

Sytuacje wyścigu mogą powodować rozmaite problemy, nawet w środowisku jednowątkowym takim jak Node.js. W niektórych przypadkach mogą wręcz prowadzić do uszkodzenia danych i są niezwykle trudne do debugowania ze względu na swoją ulotną naturę. Dlatego zawsze warto dokładnie sprawdzić potencjalne sytuacje wyścigu podczas współbieżnego wykonywania zadań.

Jednoczesne uruchamianie zbyt wielu zadań współbieżnych to zazwyczaj kiepski pomysł — bardzo szybko można doprowadzić do wykorzystania całej pamięci systemowej lub napotkać ograniczenia, takie jak maksymalna liczba otwartych deskryptorów plików. W następnym punkcie wyjaśnimy, dlaczego może to stanowić problem, a także pokażemy, jak efektywnie zarządzać liczbą zadań współbieżnych.

Ograniczone wykonywanie współbieżne

Uruchamianie zadań współbieżnych bez żadnej kontroli może łatwo doprowadzić do nadmiernego obciążenia systemu. Wyobraźmy sobie próbę jednoczesnego odczytu tysięcy plików, dostępu do wielu adresów URL lub wykonania licznych zapytań do bazy danych. Najczęstszym problemem w takich przypadkach jest wyczerpanie dostępnych zasobów. Na przykład aplikacja może próbować otworzyć zbyt wiele plików naraz, a tym samym szybko wykorzystać dostępną pulę deskryptorów plików.

Uwaga

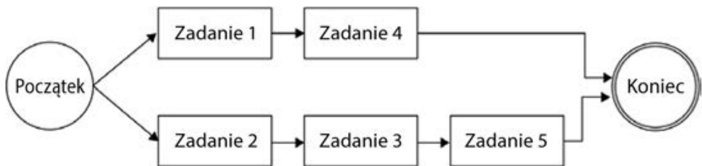
Serwer, który uruchamia nieograniczoną liczbę zadań współbieżnych w odpowiedzi na żądania użytkowników, może stać się podatny na atak typu **odmowa usługi** (ang. *denial of service*, DoS). W tego rodzaju ataku złośliwie działający podmiot może utworzyć jedno lub więcej żądań, które w jakiś sposób zmuszają serwer do wykorzystania wszystkich jego zasobów — w efekcie serwer przestanie odpowiadać na żądania użytkowników. Ograniczenie liczby jednocześnie wykonywanych zadań jest dobrą praktyką, która pomaga tworzyć aplikacje bardziej odporne na awarie.

Obecnie trzecia wersja naszego robota indeksującego strony internetowe nie nakłada żadnych ograniczeń na liczbę jednocześnie wykonywanych zadań, co powoduje, że jest on podatny na awarie w różnych scenariuszach. Na przykład jeśli uruchomimy go w dużej

witrynie, może działać przez kilka sekund, a następnie wygeneruje komunikat z błędem ECONNREFUSED. Tak się dzieje, ponieważ gdy równocześnie jest pobieranych zbyt wiele stron, serwer może zacząć odrzucać nowe połączenia z tego samego adresu IP. W takich przypadkach nasz robot ulegałby awarii i musielibyśmy ponownie uruchomić proces, aby kontynuować analizowanie danej witryny. Wprawdzie moglibyśmy dodać kod odpowiedzialny za obsługę błędu ECONNREFUSED, aby zapobiec awarii robota, ale nadal ryzykowalibyśmy przeciążenie systemu przez przydzielenie zbyt wielu jednoczesnych zadań, co prowadziłoby do innych problemów.

W tym punkcie wyjaśnimy, jak uczynić naszego robota bardziej odpornym na awarie poprzez ograniczenie współbieżności.

Diagram pokazany na rysunku 4.5 przedstawia scenariusz, w którym mamy 5 zadań do jednoczesnego wykonania, przy czym limit współbieżności wynosi 2.



Rysunek 4.5. Przykład ograniczenia współbieżności do maksymalnie dwóch jednocześnie wykonywanych zadań

Na rysunku 4.5 możemy zobaczyć, że w tym konkretnym przykładzie algorytm działa w następujący sposób:

- Na początku uruchamiamy tyle zadań, ile to możliwe, bez przekraczania limitu współbieżności. Ponieważ limit ustawiony jest na dwa, węzeł startowy uruchamia **Zadanie 1.** i **Zadanie 2**.
- Gdy tylko jedno z zadań się zakończy, uruchamiamy następne w kolejce, zawsze pamiętając o limicie. Na przykład gdy **Zadanie 2.** się kończy, uruchamiane jest **Zadanie 3**. Natomiast po zakończeniu **Zadania 1.** uruchamiane jest **Zadanie 4**. Po zakończeniu **Zadania 3.** przechodzimy do ostatniego, **Zadania 5**. Gdy **Zadanie 5.** zostanie ukończone i nie ma już więcej zadań do wykonania, cały proces dobiega końca.

W następnym podpunkcie przyjrzymy się możliwej implementacji wzorca ograniczonego wykonywania współbieżnego.

Ograniczanie współbieżności

Teraz przyjrzymy się wzorcowi, który pozwala współbieżnie wykonywać zestaw zadań z ograniczoną liczbą równoczesnych operacji:

```
const tasks = [
  // ...
]

const concurrency = 2
let running = 0
let completed = 0
```

Kup ksi k Pole ksi k

```
let nextTaskIndex = 0

function next() { // 1.
  while (running < concurrency && nextTaskIndex < tasks.length) {
    const task = tasks[nextTaskIndex++]
    task(() => { // 2.
      if (++completed === tasks.length) {
        return finish()
      }
      running--
      next()
    })
    running++
  }
}
next()

function finish() {
  // Wszystkie zadania zostały ukończone
}
```

Ten algorytm można uznać za połączenie wykonania sekwencyjnego i współbieżnego. Można zauważyć podobieństwa do obu tych wzorców:

1. Mamy funkcję `next()`, która działa jak iterator. W niej została zdefiniowana pętla, która uruchamia tyle zadań, ile to możliwe, i jednocześnie utrzymuje liczbę działających zadań w ramach ustalonego limitu współbieżności.
2. Za każdym razem gdy zadanie jest uruchamiane, przekazujemy mu funkcję wywołania zwrotnego. Decydujemy w niej, co należy zrobić po zakończeniu zadania. Jeśli zostały jeszcze jakieś zadania, wywołujemy ponownie `next()`, aby kontynuować pracę. Natomiast jeśli wszystkie zadania zostały zakończone, wywołujemy `finish()`, aby zasygnalizować, że wszystko zostało wykonane. Po drodze dbamy o prawidłową aktualizację liczników: zwiększamy liczbę ukończonych zadań i dostosowujemy liczbę działających zadań (która wzrasta za każdym razem, gdy funkcja `next()` uruchamia nowe zadanie).

Całkiem proste, prawda?

Uwaga

W tym wzorcu kontrolujemy współbieżność poprzez śledzenie za pomocą zmiennej `running` liczby zadań wykonywanych w danym momencie. Ponieważ Node.js uruchamia JavaScript w jednowątkowej pętli zdarzeń, nie ma ryzyka, że wiele zadań będzie jednocześnie modyfikować zmienną `running` — każda aktualizacja odbywa się sekwencyjnie, co eliminuje potrzebę synchronizacji. Gdy zadania się rozpoczynają, wartość `running` wzrasta, a gdy się kończą — maleje, zanim zostanie wywołana funkcja `next()`, która uruchamia kolejne zadania, jeśli to konieczne. Ponieważ wykonywanie za każdym razem pozostaje uporządkowane w ramach pętli zdarzeń, wartość `running` jest zawsze dokładna, bez konieczności stosowania blokad czy radzenia sobie z wyścigami. To kluczowa zaleta modelu współbieżności Node.js: wydajne zarządzanie zadaniami bez pułapek związanych z wielowątkowością.

Kup ksi k Pole ksi k

Globalne ograniczanie współbieżności

Nasza aplikacja robota indeksującego strony internetowe to doskonały przykład zastosowania koncepcji, które są związane z ograniczaniem liczby jednocześnie wykonywanych zadań. Bez takiego ograniczenia moglibyśmy doprowadzić do jednoczesnego analizowania tysięcy łączy, co mogłoby przeciążyć system. Poprzez kontrolowanie liczby równoczesnych pobrań możemy do procesu wprowadzić tak potrzebną przewidywalność.

Wzorzec ograniczonej współbieżności moglibyśmy zastosować do funkcji `spiderLinks()`. Jednak to ograniczyłoby liczbę zadań tylko dla łączy znalezionych na pojedynczej stronie. Na przykład gdybyśmy zdefiniowali limit współbieżności na dwa, mielibyśmy co najwyżej dwa łącza pobierane jednocześnie *z danej strony*. Skoro jednak każda strona może generować dwie kolejne operacje pobrania, ich całkowita liczba mogłaby nadal szybko rosnąć i potencjalnie wymknąć się spod kontroli.

Ogólnie rzecz biorąc, ta konkretna implementacja wzorca ograniczonej współbieżności działa dobrze, gdy mamy z góry określony zestaw zadań lub gdy zadania przyrastają w kontrolowany sposób. Jednak w przypadkach takich jak nasz program robota indeksującego strony internetowe — gdzie jedno zadanie może generować wiele nowych — nie jest on skuteczny w ograniczaniu ogólnej współbieżności. Aby to naprawić, musimy wprowadzić mechanizm, który pozwala kontrolować współbieżność na poziomie globalnym.

Kolejki na ratunek

Tak naprawdę potrzebne jest ograniczenie całkowitej liczby jednocześnie wykonywanych operacji pobierania. Dobrym sposobem na to jest wprowadzenie kolejek do zarządzania współbieżnością wielu zadań. Zobaczmy, jak to działa.

Teraz zaimplementujemy prostą klasę `TaskQueue`, łączącą kolejkę z algorytmem ograniczonej współbieżności, który właśnie omówiliśmy. Utwórzmy nowy moduł, *taskQueue.js*:

```
export class TaskQueue {
  constructor(concurrency) {
    this.concurrency = concurrency
    this.running = 0
    this.queue = []
  }

  pushTask(task) {
    this.queue.push(task)
    process.nextTick(this.next.bind(this))
    return this
  }

  next() {
    while (
      this.running < this.concurrency &&
      this.queue.length > 0
    ) {
      const task = this.queue.shift()
      task(() => {
        this.running--
        process.nextTick(this.next.bind(this))
```

```
        })
        this.running++
      }
    }
  }
```

Konstruktor tej klasy przyjmuje jako parametr wejściowy tylko limit współbieżności. Ponadto inicjalizuje zmienne egzemplarza `running` i `queue`. Pierwsza z nich to licznik używany do śledzenia wszystkich uruchomionych zadań, natomiast druga to tablica, która będzie używana jako kolejka do przechowywania zadań oczekujących.

Metoda `pushTask()` dodaje nowe zadanie do kolejki, a następnie uruchamia proces roboczy za pomocą asynchronicznego wywołania `this.next()`.

Metoda `next()` uruchamia zestaw zadań z kolejki i dba o to, aby nie przekroczyć limitu współbieżności.

Możesz zauważyć, że ta metoda ma pewne podobieństwa do wzorca przedstawionego na początku punktu o ograniczaniu współbieżności. W zasadzie uruchamia ona tyle zadań z kolejki, ile to możliwe, i nie przekracza przy tym limitu współbieżności. Gdy każde zadanie zostanie ukończone, metoda aktualizuje liczbę uruchomionych zadań, a następnie rozpoczyna kolejną rundę zadań, ponownie za pomocą asynchronicznego wywołania `next()`. Interesującą cechą klasy `TaskQueue` jest to, że pozwala dynamicznie dodawać nowe elementy do kolejki. Kolejną zaletą jest to, że mamy teraz centralny obiekt odpowiedzialny za ograniczanie współbieżności naszych zadań, który może być współdzielony między wszystkimi egzemplarzami funkcji wykonania — w naszym przypadku to `spider()`, co zobaczysz za chwilę.

Udoskonalanie kolejki zadań

Wprawdzie przedstawiona wcześniej implementacja klasy `TaskQueue` jest wystarczająca do zaprezentowania wzorca kolejki, ale aby mogła być używana w rzeczywistych projektach, potrzebuje kilku dodatkowych funkcji. Na przykład jak możemy stwierdzić, że jedno z zadań się nie powiodło? Skąd wiemy, czy wszystkie zadania w kolejce zostały zakończone?

Wróćmy do niektórych koncepcji, które omówiliśmy w poprzednim rozdziale, i przekształćmy `TaskQueue` w klasę `EventEmitter`, abyśmy mogli emitować zdarzenia, które będą informować o niepowodzeniach zadań oraz powiadamiać obserwatory, gdy kolejka jest pusta.

Pierwszą zmianą, którą trzeba wprowadzić, jest zaimportowanie klasy `EventEmitter` i rozszerzenie klasy `TaskQueue` o niezbędną funkcjonalność:

```
import { EventEmitter } from 'node:events'

export class TaskQueue extends EventEmitter {
  constructor (concurrency) {
    super()
    // ...
  }
  // ...
}
```

Kup książkę Poleć książkę

W tym momencie możemy użyć `this.emit` do wyzwalania zdarzeń w metodzie `next()` klasy `TaskQueue`:

```
next () {
  if (this.running === 0 && this.queue.length === 0) { // 1.
    return this.emit('empty')
  }

  while (this.running < this.concurrency && this.queue.length) {
    const task = this.queue.shift()
    task((err) => { // 2.
      if (err) {
        this.emit('error', err)
      }
      this.running--
      process.nextTick(this.next.bind(this))
    })
    this.running++
  }
}
```

Porównując tę implementację z poprzednią, można zauważyć dwa nowe elementy:

1. Przy każdym wywołaniu funkcji `next()` sprawdzamy, czy żadne zadanie nie jest aktualnie wykonywane oraz czy kolejka jest pusta. W takim przypadku to oznacza, że kolejka została opróżniona i możemy wywołać zdarzenie pustej kolejki, `empty`.
2. Funkcja wywołania zwrotnego zakończenia każdego zadania może być teraz uruchamiana poprzez przekazanie błędu. Sprawdzamy, czy faktycznie został przekazany błąd, co wskazuje na niepowodzenie zadania, a następnie propagujemy go za pomocą zdarzenia błędu, `error`.

Zwróć uwagę, że w przypadku wystąpienia błędu celowo utrzymujemy działanie kolejki. Nie przerywamy innych trwających zadań ani nie usuwamy żadnych oczekujących. To jest dość powszechne w systemach opartych na kolejkach. Błędy są czymś spodziewanym i zamiast pozwalać systemowi się zawiesić w takich sytuacjach, lepiej jest je zidentyfikować i pomyśleć o strategiach ponownych prób lub odzyskiwania. Te koncepcje omówimy nieco szerzej w rozdziale 13., poświęconym wzorcom komunikacji i integracji.

Uwaga

Hipotetycznie, jeśli chcesz elegancko zatrzymać wszystkie zadania po napotkaniu błędu, możesz rozważyć wprowadzenie mechanizmu sygnalizującego kolejce, aby przestała przyjmować i przetwarzać nowe zadania. Wyobraź sobie scenariusz, w którym dodajemy flagę `this.stopped` do klasy `TaskQueue`. W przypadku błędu ta flaga natychmiast otrzymuje wartość `true`, a zawartość kolejki jest usuwana. Takie podejście zapobiegłoby uruchamianiu następnych zadań oczekujących w kolejce. Moglibyśmy również dodać logikę uniemożliwiającą dodawanie do kolejki nowych zadań (np. przez zgłaszanie błędu w metodzie `pushTask()`, gdy wartość flagi `this.stopped` wynosi `true`). Choć takie podejście zapobiegłoby uruchamianiu nowych zadań przez kolejkę, warto zauważyć, że nie byłoby w stanie zatrzymać tych zadań, które już są w trakcie wykonywania lub oczekują na zakończenie jakiegoś asynchronicznego wywołania.

Kup ksi k Pole ksi k

Wersja czwarta robota indeksującego strony internetowe

Skoro mamy już ogólną kolejkę do wykonywania zadań w ograniczonym przepływie współbieżnym, użyjmy jej od razu do refaktoryzacji aplikacji robota indeksującego strony internetowe.

Wykorzystamy egzemplarz `TaskQueue` jako bufor zadań, a każdy adres URL, który chcemy przeanalizować, musi zostać dodany do kolejki jako zadanie. Początkowy adres URL zostanie dodany jako pierwsze zadanie, każdy inny adres URL odkryty podczas procesu indeksowania również zostanie dodany. Kolejka zajmie się obsługą harmonogramu i będzie dbała o to, aby liczba zadań w toku (czyli liczba stron pobieranych lub odczytywanych z systemu plików) w żadnym momencie nie przekraczała limitu współbieżności skonfigurowanego dla danego egzemplarza `TaskQueue`.

W funkcji `spider()` zdefiniowaliśmy już logikę indeksowania danego adresu URL. Możemy to uznać za ogólne zadanie indeksowania. Dla zapewnienia większej przejrzystości najlepiej będzie zmienić nazwę tej funkcji na `spiderTask`:

```
function spiderTask(url, maxDepth, queue, cb) { // 1.
  const filename= urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    if (err) {
      // Błąd podczas sprawdzania pliku
      return cb(err)
    }
    if (alreadyExists) {
      if (!filename.endsWith('.html')) {
        // Pomijanie zasobów innych niż HTML
        return cb()
      }
      return readFile(filename, 'utf8', (err, fileContent) => {
        if (err) {
          // Błąd podczas odczytu pliku
          return cb(err)
        }
        spiderLinks(url, fileContent, maxDepth, queue) // 2.
      return cb()
    })
  }
  // Plik nie istnieje, pobierz go
  download(url, filename, (err, fileContent) => {
    if (err) {
      // Błąd podczas pobierania pliku
      return cb(err)
    }
    // W przypadku pliku HTML należy go przeanalizować
    if (filename.endsWith('.html')) {
      spiderLinks(url, fileContent.toString('utf8'), maxDepth, queue) // 2.
      return cb()
    }
    // W przeciwnym razie zakończ działanie
    return cb()
  })
 })
}
```

Kup książkę Poleć książkę

Oprócz zmiany nazwy funkcji wprowadziliśmy także kilka innych drobnych modyfikacji:

1. Sygnatura funkcji przyjmuje teraz nowy parametr, `queue`. Jest to egzemplarz `TaskQueue`, którą musimy przekazać, aby w razie potrzeby móc dodawać nowe zadania.
2. Funkcja odpowiedzialna za dodawanie nowych łączy do przeanalizowania to `spiderLinks()`, więc musi zostać przekazany egzemplarz kolejki podczas wywoływania tej funkcji po pobraniu nowej strony.

Za chwilę zobaczysz, że funkcja `spiderLinks()` zostanie uproszczona do postaci funkcji synchronicznej, ponieważ będzie musiała jedynie dodawać zadania do kolejki. Nie będziemy już musieli przekazywać do niej wywołania zwrotnego i zamiast tego po prostu wywołamy polecenie `return cb()` po jej wykonaniu.

Przyjrzyjmy się teraz funkcji `spiderLinks()`. Można ją znacznie uprościć, ponieważ nie musi już śledzić ukończenia zadań — teraz zajmuje się tym kolejka. Jej wykonanie stanie się w zasadzie synchroniczne. Wystarczy, że wywoła funkcję `spider()`, którą zdefiniujemy za chwilę, aby dodać nowe zadanie do kolejki dla każdego odkrytego łącza:

```
function spiderLinks(currentUrl, body, maxDepth, queue) {
  if (maxDepth === 0) {
    return
  }

  const links = getPageLinks(currentUrl, body)
  if (links.length === 0) {
    return
  }

  for (const link of links) {
    spider(link, maxDepth - 1, queue)
  }
}
```

Wróćmy teraz do funkcji `spider()`, która ma służyć jako *punkt wejścia* dla pierwszego adresu URL. Będzie ona również używana w celu dodawania każdego nowo odkrytego adresu URL do kolejki:

```
const spidering = new Set() // 1.
export function spider(url, maxDepth, queue) {
  if (spidering.has(url)) {
    return
  }
  spidering.add(url)
  queue.pushTask(done => { // 2.
    spiderTask(url, maxDepth, queue, done)
  })
}
```

Jak widać, ta funkcja ma teraz dwa główne zadania:

1. Za pomocą zbioru `spidering` zarządza rejestrem już odwiedzonych lub przetwarzanych adresów URL.
2. Dodaje nowe zadanie do kolejki. Po wykonaniu tego zadania zostanie wywołana funkcja `spiderTask()`, co praktycznie rozpoczyna indeksowanie danego adresu URL.

Na koniec możemy zaktualizować skrypt *spider-cli.js*, który pozwala uruchomić robota z poziomu wiersza poleceń:

```
import { spider } from './spider.js'
import { TaskQueue } from './TaskQueue.js'

const url = process.argv[2] // 1.
const maxDepth = Number.parseInt(process.argv[3], 10) || 1
const concurrency = Number.parseInt(process.argv[4], 10) || 2

const spiderQueue = new TaskQueue(concurrency) // 2.
spiderQueue.on('error', console.error)
spiderQueue.on('empty', () => console.log('Pobieranie zakończone'))

spider(url, maxDepth, spiderQueue) // 3.
```

Ten skrypt składa się teraz z trzech głównych części:

1. Przetwarzanie argumentów wiersza poleceń. Warto zauważyć, że skrypt przyjmuje trzeci dodatkowy parametr, który można wykorzystać w celu dostosowania poziomu współbieżności.
2. Tworzymy obiekt `TaskQueue` i dołączamy do niego komponenty nasłuchujące zdarzeń `error` i `empty`. Gdy wystąpi błąd, chcemy go po prostu wyświetlić. Natomiast kiedy kolejka jest pusta, oznacza to, że zakończyliśmy przeszukiwanie witryny.
3. Na koniec uruchamiamy proces przeszukiwania poprzez wywołanie funkcji `spider`.

Po wprowadzeniu tych zmian możemy ponownie uruchomić moduł robota indeksującego strony internetowe. Jeśli wydasz następujące polecenie:

```
$ node spider-cli.js https://loige.co 1 4
```

zauważysz, że jednocześnie aktywne będą nie więcej niż cztery operacje pobierania.

Na tym ostatnim przykładzie kończymy omówienie wzorców opartych na funkcjach wywołań zwrotnych.

Podsumowanie

Na początku tego rozdziału wspomnieliśmy, że Node.js może być trudny do opanowania ze względu na swoją asynchroniczną naturę, szczególnie dla programistów przyzwyczajonych do innych platform. Jednak, jak się przekonałeś, API asynchroniczne działa na Twoją korzyść. Narzędzia, które poznałeś, są elastyczne i oferują solidne rozwiązania wielu typowych problemów, a jednocześnie pozwalają stosować różne style programowania, dostosowane do indywidualnych preferencji.

W tym rozdziale kontynuowaliśmy także refaktoryzację i ulepszanie przykładu robota indeksującego strony internetowe. Praca z kodem asynchronicznym czasem wymaga znalezienia odpowiedniej równowagi między prostotą i efektywnością. Dlatego warto poświęcić nieco czasu na przyswojenie omówionych tutaj koncepcji i eksperymentowanie z nimi.

Nasza przygoda z programowaniem asynchronicznym na platformie Node.js dopiero się zaczyna. W kolejnych rozdziałach poznasz inne powszechnie stosowane techniki, takie jak obietnice i mechanizm `async/await`. Gdy zaznajomisz się ze wszystkimi tymi podejściami, będziesz w stanie wybrać najlepsze rozwiązanie dla swoich potrzeb — a nawet łączyć różne techniki w ramach jednego projektu.

Zanim przejdziesz dalej, gorąco zachęcamy do wykonania ćwiczeń, gdyż pozwolą Ci ugruntować wiedzę. Pomogą one utrwalić kluczowe koncepcje i przygotują Cię do lektury kolejnych rozdziałów.

Ćwiczenia

1. **Konkatenacja plików.** Zaimplementuj funkcję `concatFiles()`, która działa w stylu wywołania zwrotnego i przyjmuje dwie lub więcej ścieżek dostępu do plików tekstowych w systemie plików oraz ścieżkę dostępu do pliku docelowego:

   ```
   functiona concatFiles (plikZrodlowy1, plikZrodlowy2, plikZrodlowy3, ...,
                          plikDocelowy, funkcjaWywolaniaZwrotnego) {
     //…
   }
   ```

   Ta funkcja musi skopiować zawartość każdego pliku źródłowego do pliku docelowego, zachowując ich kolejność zgodnie z listą argumentów. Na przykład w przypadku dwóch plików, jeśli pierwszy zawiera tekst *foo*, a drugi *bar*, wówczas w pliku docelowym funkcja powinna zapisać ciąg tekstowy *foobar* (a nie *barfoo*). Warto zauważyć, że podana przykładowa sygnatura funkcji nie jest poprawną składnią JavaScriptu — należy znaleźć inny sposób obsługi dowolnej liczby argumentów. Można na przykład skorzystać ze składni tzw. **parametrów resztowych** (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters*).

2. **Rekurencyjne wyświetlanie listy plików.** Zdefiniuj funkcję `listNestedFiles()` w stylu wywołania zwrotnego, która jako dane wejściowe przyjmuje ścieżkę dostępu do katalogu w lokalnym systemie plików i asynchronicznie przeszukuje wszystkie podkatalogi, aby ostatecznie zwrócić listę znalezionych plików. Oto jak powinna przedstawiać się sygnatura tej funkcji:

   ```
   function listNestedFiles (katalog, funkcja_wywołania_zwrotnego) { /* ... */ }
   ```

   Dodatkowe punkty można zdobyć za uniknięcie piekła wywołań zwrotnych. Możesz swobodnie tworzyć dodatkowe funkcje pomocnicze, jeśli to konieczne.

3. **Wyszukiwanie rekurencyjne.** Zdefiniuj funkcję `recursiveFind()`, która działa w stylu wywołania zwrotnego i przyjmuje ścieżkę dostępu do katalogu w lokalnym systemie plików i słowo kluczowe, zgodnie z następującą sygnaturą:

   ```
   function recursiveFind(katalog, słowo_kluczowe, funkcja_wywołania_zwrotnego)
   ↪{ /* ... */ }
   ```

   Jej zadaniem jest znalezienie w podanym katalogu wszystkich plików tekstowych, które zawierają w swojej treści wskazane słowo kluczowe. Lista pasujących plików powinna zostać zwrócona za pomocą funkcji wywołania

Kup ksi k Pole ksi k

zwrotnego po zakończeniu operacji wyszukiwania. Jeśli nie znaleziono żadnego dopasowanego pliku, funkcja wywołania zwrotnego musi zostać wykonana z pustą tablicą. Na przykład jeśli masz pliki *foo.txt*, *bar.txt* i *baz.txt* w katalogu *myDir*, a podane słowo kluczowe `batman` znajduje się w plikach *foo.txt* i *baz.txt*, to musisz mieć możliwość wykonania następującego polecenia:

```
recursiveFind('myDir', 'batman', console.log)
// Dane wyjściowe powinny mieć postać ['foo.txt', 'baz.txt']
```

Dodatkowe punkty możesz zdobyć, jeśli skorzystasz z wyszukiwania rekurencyjnego (sprawdzenie plików tekstowych we wszystkich podkatalogach). Ekstra dodatkowe punkty, jeśli uda Ci się przeprowadzić wyszukiwanie w różnych plikach i podkatalogach jednocześnie — uważaj jednak, aby kontrolować liczbę równoległych zadań!

4. **Narzędzie do sprawdzania nieprawidłowych łączy.** Zdefiniuj funkcję `checkBrokenLinks()`, która przyjmuje adres URL i maksymalny poziom zagłębienia, a następnie dostarcza informacje o wszystkich nieprawidłowych łączach (czyli zwracających kod stanu 404), które napotka podczas procesu przetwarzania strony internetowej. Pracę możesz rozpocząć od kodu, który wcześniej utworzyliśmy dla robota indeksującego strony internetowe, a następnie zmodyfikuj go tak, aby zamiast pobierać zawartość stron, sprawdzał tylko kod stanu HTTP każdego łącza. Jeśli tym kodem jest 404, funkcja powinna wyświetlić adres URL nieprawidłowego łącza i kontynuować sprawdzanie kolejnych łączy. Wskazówka: podczas sprawdzania łączy spróbuj użyć metody `HEAD` zamiast `GET`, ponieważ pobiera ona tylko nagłówki, co może przyspieszyć proces i zmniejszyć zużycie przepustowości.

Kup ksi k Pole ksi k

Skorowidz

A

abstrakcja
  odpowiedzi, 669, 674
  żądania, 668, 672
Adapter
  schemat wzorca, 329
  w Node.js, 332
Adres zwrotny, 671
aktywne oczekiwanie, 33
algorytm obliczania sumy podzbioru, 530–533
AMQP, 632
  adres zwrotny, 671
  kolejka, 633
  konkurujące konsumenty, 656
  wiązanie, 633
  wymiana, 633
antywzorzec, 107, 184
anulowanie operacji asynchronicznych, 522
API
  asynchroniczne, 100, 104
  buforowania, 196
  obietnicy, 160
  oparte na buforach, 198
  proxy, 600
  synchroniczne, 100, 502
  systemu plików, 329
aplikacja
  czatu, 618, 641
  do łamania skrótów, 648, 657, 661
  internetowa
    testy E2E, 478
    testy integracyjne, 468
architektura
  mikrousług, 596, 597
    wzorce integracji, 600
  monolityczna, 595, 596
  równorzędna P2P, 627, 647
  wielousługowa, 575
asercje asynchroniczne, 491
asynchroniczna inicjalizacja komponentów, 502
  kolejkowanie wywołań metod, 507
  kontrola lokalna, 505
  opóźnione uruchomienie, 505
  wzorzec Stan, 509
asynchroniczne
  API, 100, 104
  asercje, 491
  funkcje, 94–98, 102, 178, 523, 526
  generatory, 379
  iteratory, 209, 375, 380
  obiekty iterowalne, 376
  operacje
    anulowanie, 522
  przekazywanie komunikatów, 613
  przetwarzanie wsadowe, 512
  wywołania, 524
  wzorce kontroli, 122
  wzorce kontroli przepływu, 132, 155, 237
  zadania, 139
  zdarzenia, 114
atak
  CSRF, 384
  DDoS, 533
  DoS, 136, 533
atrapa, 406, 443

B

baza danych, 461
BDD, behavior-driven development, 403
bezpieczeństwo typów, 77
biblioteka
  assert, 417
  Bluebird, 522
  commander.js, 287
  delegates, 310
  indexed-string-variation, 651
  Level, 329
  libuv, 38
  MobX, 319

biblioteka
  ORM, 286
  p-lazy, 177
  Puppeteer, 483
  Undici, 448
  workerpool, 546
  XState, 353
  ZeroMQ, 353, 626
błędy
  CPS, 105
  propagowanie, 105, 111
broker komunikatów, 604, 616, 656, 657
  dystrybucja zadań, 657
  Redis, 623
Budowniczy, 278
  konstruktor obiektów URL, 282
  tworzenie obiektów, 278
buforowanie, 194, 202
  z użyciem obietnic, 519
  zapytań, 521
  żądań asynchronicznych, 514

C

ciągła integracja, CI, 409
ciągłe
  dostarczanie, CD, 409
  wdrażanie, CD, 409
CommonJS, 41, 78
CWE, Common Weakness Enumeration, 77
czat, 618

D

definicja modułów
  asynchroniczna, AMD, 48
  uniwersalna, UMD, 48
Dekorator, 328
  implementacja wzorca, 320
  schemat działania, 320
  w Node.js, 326
dekoratory
  ECMAScript, 327
  w TypeScripcie, 327, 328
dekorowanie
  bazy danych Level, 324
  obiektów
    poprzez monkey patching, 321
    za pomocą kompozycji, 320
    za pomocą obiektu Proxy, 322
demultipleksacja zdarzeń, 34
demultipleksowanie, 34, 254, 257, 259
deszyfrowanie, 205
DI, dependency injection, 458
Docker, 586
domknięcie, closure, 92
DoS, denial of service, 533
dostarczanie typu push i pull, 612
drzewo, 651
dublery testowe, 406
dynamiczne rozdzielanie obciążenia, 575
dystrybucja zadań, 645, 657
  wzorce, 647
  za pomocą strumieni Redisa, 660
działanie Node.js, 31

E

ECMAScript, 40
efektywność
  czasowa, 197
  przestrzenna, 196
eksportowanie
  domyślne, 55, 57
  mieszane, 56
  nazwane, 53, 57
emiter zdarzeń, 116
Express
  oprogramowanie pośredniczące, 384

F

Fabryka, 272
  hermetyzacja, 274
  implementacja obiektów, 273
  profiler kodu, 275
  tworzenie obiektów, 273
  w Node.js, 278, 285
framework
  Angular, 327
  Ava, 420
  Cypress, 483
  Express, 384
  Jasmine, 420
  Jest, 420
  Kubernetes, 589
  LoopBack, 319
  Mocha, 420
  NestJS, 327
  Vitest, 420
  Vue.js, 319
funkcja setImmediate, 102, 104, 533

funkcje
asynchroniczne, 95–97, 178
anulowanie działania, 523, 526
dołączone, 394
generujące, 371
niespójne, 98
opakowujące, 524
samowywołujące, 50
skrótu, 649
strzałki, 323
synchroniczne, 93, 97
testowe, 421
wykonawcze, 162, 173
wywołań zwrotnych, 36, 91, 100, 104, 116

G

generator sum kontrolnych, 251
generatory, 371
asynchroniczne, 379
sterowanie iteratorem, 372
używanie, 374
gniazda
PUB i SUB, 627
PUSH i PULL, 648
gorliwa ewaluacja, 370
grupowanie zapytań, 521
grupy konsumentów Redisa, 660

H

hermetyzacja, 274

I

idempotentność, 561
Identyfikator korelacji, 666
wymiana komunikatów, 667
identyfikatory modułów, 58
imitacja, 406, 442
importowanie
domyślne, 55
plików JSON, 85
przestrzeni nazw, 54
statyczne i dynamiczne, 58
indeksowanie stron, 123, 134, 140, 150, 169, 171
inicjalizacja asynchroniczna, 503
input/output, I/O, 31
IntelliSense, 78
interfejs
AbortController, 526
powiadamiania o zdarzeniach, 34
internet rzeczy, IoT, 42
iteracja, 165, 183
sekwencyjna, 133
Iterator, 357
implementacja protokołu iterowalnego, 364
protokół iteratora, 357
protokół iterowalny, 360
w Node.js, 383
iteratory, 362, 365
asynchroniczne, 209, 375, 380

J

JavaScript, 39
język
JavaScript, 39
SQL, 286
TypeScript, 43
JSON, 85

K

kacze typowanie, 277
klasa
AbortController, 526
Duplex, 220
EventEmitter, 109
metody, 109
tworzenie egzemplarza, 110
PassThrough, 228
Readable, 207
TaskQueue, 169
Transform, 221
Url, 282
Writable, 214
klonowanie, 552
kolejka, queue, 633
komunikatów, MQ, 630, 641
zadań, 147, 148
kompilator TypeScriptu, 87
komponenty
inicjalizowane asynchronicznie, 502
nasłuchujące, 109, 113
Node.js, 39
kompozycja obiektów, 308
kompresja gzip, 196, 197
komunikacja
asynchroniczna, 613
bezpośrednia, 615
dostarczanie komunikatu, 630
dostarczanie przez wypychanie, 612
kolejka komunikatów, 614, 630

komunikacja
oparta na AMQP, 632
pobieranie na żądanie, 612
przesyłanie komunikatów, 640
punkt-punkt, 656, 667
strumień komunikatów, 615, 640
synchroniczna, 613
typy komunikatów, 610
wzorce jednokierunkowe, 609
z pośrednikiem, 615
żądanie-odpowiedź, 609, 672
komunikat
dokumentu, 611
polecenia, 610
zdarzenia, 611
konstrukcja async/await, 180, 183, 184
Konstruktor ujawniający, 288
niemodyfikowalny bufor, 289
w Node.js, 291
kontener
Dockera, 586
linuksowy, 586
kontrola inicjalizacji, 505
Kubernetes, 589, 590
wdrażanie i skalowanie aplikacji, 591

L

Level, 324, 329, 332
implementacja wtyczki, 324
LevelDB, 324
lokalizatory, 490

Ł

łamanie skrótów, 648, 657, 661
łańcuch obietnic, 158, 187

M

mapowanie obiektowo-relacyjne, ORM, 511
menedżer
konfiguracji, 354
oprogramowania pośredniczącego, 387
pakietów
npm, 28
pnpm, 28
yarn, 28
metoda Array.forEach, 184
metody
pułapek, trap methods, 312
szablonowe, 353
mikrousługi, 596
mnożenie macierzy, 336
moduł, 28, 47, 48
child_process, 42
cluster, 554, 558
działanie, 554
odporność na awarie, 558
skalowanie, 557
crypto, 42
dgram, 42
fs, 42
http, 42
https, 42
net, 42
v8, 42
vm, 42
moduły
algorytm rozwiązywania, 61
CommonJS, 78, 80
importowanie modułów ES, 84
definiowanie systemu, 48
ES w Node.js, 52
ES, 51
brakujące odwołania, 81
eksportowanie i importowanie, 53, 55, 58
importowanie modułów CommonJS, 82
składnia, 53
tryb ścisły, 80
format wyjściowy, 89
identyfikatory, 58
łączenie, 295
modyfikujące inne moduły, 71
pakietów, 61
pakowanie, 49
plikowe, 61
podstawowe Node.js, 61
użytkownika, 42
w TypeScripcie, 87
wczytywanie, 63
zalety stosowania, 48
zależności cykliczne, 65, 67
monitorowanie adresów URL, 241
monkey patching, 71, 76, 77, 311, 321
multipleksowanie, 34, 254, 255, 259

N

narzędzia
do iteracji, 365
do iteracji asynchronicznej, 381
do raportowania testów, 432
do testowania, 421

Kup książkę Poleć książkę

narzędzie
  Ansible, 573
  Autocannon, 518
  Biome, 57
  Browserify, 49
  CLI, 27
  Consul, 576
  createRequire, 86
  Docker, 586
  Knex, 278
  npm, 45
  Packer, 573
  pakowacz modułów, 41
  Playwright, 485
  polyfill, wypełniacz, 40
  Redis, 623
  Selenium, 483
  Terraform, 573
  transpilator, 40
  ts-node, 44
  tsx, 44
  Webpack, 49, 88
Node.js, 26, 27
notacja #, 176

O

obiekt Proxy, 312, 317
obiekty
  iterowalne, 362, 365
  konfiguracyjne, 338
obietnice, promises, 91, 156, 519
  API, 160
  leniwe, lazy, 162, 173
  obiekty podobne, 159
  przepływ współbieżny, 168
  specyfikacja Promises/A+, 159
  tworzenie, 162
obserwator, observer, 108
Obserwator zmian, 317
  implementacja wzorca, 317
  w Node.js, 319
obsługa
  błędów, 180, 234, 235
  komunikacji stanowej, 563
odporność, 558
odroczone wykonanie, 102
odwrotny serwer proxy, 567–569, 572, 575
odwrócenie sterowania, IoC, 302
opakowywanie wywołań asynchronicznych, 524
operacje wejścia-wyjścia, 31
  blokowanie, 103
  blokujące, 32
  nieblokujące, 33
oprogramowanie pośredniczące, 384
  dla ZeroMQ, 386
  menedżer, 387
  w Expressie, 384
  wdrażanie, 389
  wykorzystanie infrastruktury, 390
ORM, object-relational mapping, 511

P

P2P, peer-to-peer, 626, 647
pakiet
  @types/node, 45
  amqplib, 635
  npm, 576
  pg, 512
pakowacz modułów, 41
Pełnomocnik, 317, 328
  kompozycja obiektów, 308
  metody implementacji, 306
  obiekt Proxy, 312
  rozszerzanie obiektów, 311
  schemat wzorca, 305
  strumień typu Writable, 315
  w Node.js, 319
pętla
  for-of, 362
  zdarzeń, 35, 96, 103
piekło wywołań zwrotnych, callback hell, 127
piramida
  testów, 415
  zagłady, pyramid of doom, 127
platforma Node.js, 26
Playwright
  API, 487
  inicjalizacja projektu, 486
  limity czasowe, 492
  symulacja zachowania użytkownika, 493
  tworzenie testu E2E, 485
pliki JSON, 85
podtesty, 424
pokrycie kodu, 405
Polecenie, 393, 507
  elementy składowe wzorca, 393
  implementacja wzorca, 394
  w Node.js, 398
polyfill, 40, 314

połączenie WebSocket, 667
ponowne uruchomienie bez przestoju, 561
Pośrednik, 385, *Patrz także* oprogramowanie pośredniczące
  struktura wzorca, 385
  w Node.js, 392
potoki, 233, 234
potokowanie, 247
problem sumy podzbioru, 529
procesy
  potomne, 536
  zewnętrzne, 536
programowanie
  asynchroniczne, 91, 94, 123
  funkcyjne, 260
  sterowane testami, TDD, 403, 407
  sterowane zachowaniem, BDD, 403, 408
  synchroniczne, 91, 93
  ze strumieniami, 192
  zorientowane obiektowo, 109
Promises/A+, 159
promisification, 163
propagowanie błędów, 105, 111
protokół
  iteratora, 357
  iterowalny, 360
    implementacja, 364
  komunikacyjny
    AMQP, 616, 632
    MQTT, 616
przeglądarka internetowa, 483
przełączanie pakietów, 255
przeplatanie
  długotrwałych zadań, 535
  kroków algorytmu, 533
przepływ sterowania, 132
przestrzeń użytkownika, 28
przetwarzanie wsadowe
  asynchroniczne, 512
  z użyciem obietnic, 519
  żądań, 519
przygotowanie – działanie – asercja, AAA, 405
przypadki testowe, 426
Publikowanie-Subskrybowanie, 617
  broker komunikatów, 623
  kolejkowanie komunikatów, 630
  strumienie komunikatów, 641
  typu P2P, 626
pułapki, 312, 314

R

raportowanie testów
  dot, 433
  junit, 433
  lcov, 433
  spec, 432
  tap, 432
rdzeń Node.js, 28
reaktor, 36
reguła KISS, 123
rejestr usług, 574, 575
robot indeksujący strony, 123, 134, 140, 150, 169, 171
rozszerzanie obiektów, 311
rozwiązywanie
  modułów, 61
  obietnic, 187
równoległość, 138
równoważenie obciążenia, 552, 577
  między serwerami, 584
  między węzłami równorzędnymi, 582
  moduł cluster, 554
  obsługa komunikacji stanowej, 563
  odwrotny serwer proxy, 567
RPC, remote procedure calls, 611

S

scalanie plików tekstowych, 253
sekwencyjne
  przeglądanie łączy, 135
  wykonywanie zadań, 132
serwer
  API, 516
  czatu, 627, 639
  HTTP, 555
  proxy, 569
  WWW, 569
silnik
  V8, 39
  wejścia-wyjścia, 38
Singleton, 292
  zależności między modułami, 296
skalowanie aplikacji, 549
  dekompozycja według funkcjonalności, 551, 594
  klonowanie, 551, 552
  mikrousługi, 596, 599
  odwrotny serwer proxy, 567
  partycjonowanie danych, 551
  pionowe, 552

Kup ksi k Pole ksi k

poziome, 553
poziome dynamiczne, 574
przy użyciu kontenerów, 586
rozkładanie obciążenia, 550
równoważenie obciążenia, 552, 554
w środowisku Kubernetes, 591
za pomocą modułu cluster, 554, 557
SLA, service level agreement, 561
słowo kluczowe
async, 177, 183, 379
await, 80, 177, 183
export, 53
import, 54
require, 41
this, 81
SQL, 286
SQLite, 296
Stan, 344
gniazdo klienta TCP, 346
kolejkowany, queuing state, 509
schemat wzorca, 344
w Node.js, 353, 511
zainicjalizowany, initialized state, 509
zastosowanie, 345
Strategia, 336
obiekty konfiguracyjne, 338
struktura wzorca, 337
w Node.js, 344
strumienie, 194, 206
implementacja łączenia, 249
internetowe, 263
konwersja strumieni Node.js, 264
leniwe, 232
łączenie, 203, 233, 247
Node.js, 380
konwersja strumieni internetowych, 265
obiektów
demultipleksowanie, 259
multipleksowanie, 259
odczytu, 260
filtrowanie, 261
iteracja, 261
mapowanie, 260
ocena, 261
ograniczanie, 261
przekształcanie, 260
redukowanie, 261
wyszukiwanie, 261
pobieranie całej zawartości, 266
Redisa, 641, 660, 661
rozwidlanie, 251
scalanie, 252
współbieżne
implementacja, 240
wykonywanie sekwencyjne, 237
wykonywanie współbieżne, 239
strumieniowanie, 194, 202
strumień
klasy Readable, 207
komunikatów, 641
typu Duplex, 220
typu PassThrough, 228
obserwowalność, 228
późne łączenie, 229
typu Readable
implementacja, 209
odczytywanie ze strumienia, 207
typu Transform, 221
agregowanie danych, 225
filtrowanie danych, 225
implementacja, 222
typu Writable, 315
ciśnienie zwrotne, 216
implementacja, 218
zapis do strumienia, 214
styl przekazywania kontynuacji, CPS, 93, 105, 122
asynchroniczny, 94
synchroniczny, 93
symulowanie żądań HTTP, 444
synchroniczna funkcja, 93, 97, 98
synchroniczne
API, 100, 502
zdarzenia, 114
synchroniczny demultiplekser zdarzeń, 34
system modułowy, 41, 47, 48
CommonJS, 47, 78
ECMAScript, 47, 51
w JavaScripcie, 48
Szablon, 353
diagram UML, 354
menedżer konfiguracji, 354
w Node.js, 357
sześcian skalowania, scale cube, 550, 552
szpieg, 406, 442
szybkie niepowodzenie, 108
szyfrowanie, 203

Ś

ścisłe powiązanie, 596

T

tablice pośrednie, 370
TDD, test-driven development, 403
test
    E2E, 413
        aplikacji internetowej, 478
        symulacja zachowania użytkownika, 483
        tworzenie, 477, 485
    integracyjny, 412
        aplikacji internetowej, 468
        bazy danych, 461
        tworzenie, 461
    jednostkowy, 411
        imitowanie modułów, 450
        imitowanie operacji importowania, 457
        imitowanie zależności, 453
        kodu asynchronicznego, 437
        symulowanie żądań HTTP, 444, 448
        tworzenie, 417, 436
        tworzenie szpiegów, 442
        wstrzykiwanie zależności, 458
tester
    tryb obserwacji, 429
    ukierunkowane wykonywanie testów, 430
testowanie oprogramowania, 403
testowany system, SUT, 404
testy
    frameworki, 420
    informacje o stopniu pokrycia kodu, 433
    narzędzie w TypeScripcie, 435
    organizacja, 422
    piramida testów, 415
    program uruchamiający, 420
    przypadki testowe, 426
    raportowanie, 432
    rodzaje testów, 415
    w Node.js, 420
    wbudowane narzędzie, 421
    zestawy, 428
transpilacja, 314
transpilator, 40
tworzenie
    aplikacji czatu, 618
    atrap, 443
    gniazda klienta TCP, 346
    klienta HTTP, 584
    kontenera Dockera, 586
    niemodyfikowalnego bufora, 289
    obietnicy, 162
    profilera kodu, 275
    robota indeksującego, 123
    serwera HTTP, 555
    strumienia zapisującego, 315
    szpiegów, 442
    testów, 417
        E2E, 477, 485
        integracyjnych, 461
        jednostkowych, 417, 436
TypeScript, 43
    generowanie danych wyjściowych, 89
    kompilator, 87
    konfiguracja formatu wyjściowego, 89
    moduły, 87
    rozwiązywanie modułów, 90

U

uwolnienie Zalgo, 98

V

V8, 39

W

wątki robocze, worker threads, 543
wczytywanie modułów, 63
WebAssembly, Wasm, 43
wektor inicjujący, initialization vector, 204
wiązanie, binding, 39, 633
    tylko do odczytu, 64
wielozadaniowość
    kooperatywna, 524
    z wywłaszczaniem, 524
współbieżne wykonywanie zadań, 138, 139, 168, 185
    ograniczenia, 144, 169, 186, 244
    uporządkowanie, 245
współbieżność, 138
    ograniczanie, 145, 169, 186, 244
    ograniczanie globalne, 147
    uporządkowana, 245
Wstrzykiwanie zależności, DI, 458
    między modułami, 298
    przez funkcję, 301
    przez konstruktora, 301
    przez właściwości, 301
wyciek pamięci, 113, 188
wyjątki, 106
wykonywanie
    równoległe, 138
    sekwencyjne, 132, 165, 183, 184, 237
    współbieżne, 138, 139, 168, 185, 239
        ograniczone, 144, 169, 186, 244
        uporządkowane, 245

wymiana, exchange, 633, 657
wyrażenie await, 178
  na najwyższym poziomie, 179
wyścig, 142, 143
wywołania zwrotne, callbacks, 36, 91, 92, 105, 127
  antywzorce, 127
  dyscyplina, 129
  łączenie ze zdarzeniami, 117
  najlepsze praktyki, 128
  niesynchroniczne, 97
  zasady, 130
wzorce
  żądanie-odpowiedź, 609
  dystrybucji zadań, 645
    fan-in, 647
    fan-out, 647
  jednokierunkowe, 609
  komunikacji i integracji, 608
    Publikowanie-Subskrybowanie, 617
  kontroli przepływu, 155, 237
  potokowania, 247
  projektowe behawioralne, 335
    Iterator, 357
    Polecenie, 393
    Pośrednik, 383
    Stan, 344
    Strategia, 336
    Szablon, 353
  projektowe konstrukcyjne, 272
    Budowniczy, 278
    Fabryka, 272
    Konstruktor ujawniający, 288
    Singleton, 292
    Wstrzykiwanie zależności, 298, 458
  projektowe strukturalne, 305
    Adapter, 329
    Dekorator, 319
    Obserwator zmian, 317
    Pełnomocnik, 305
  przepływu sterowania, 132
  żądań i odpowiedzi
    Adres zwrotny, 671
    Identyfikator korelacji, 666
wzorzec
  agregacja strumieniowa, 228
  dla przepływu wykonania współbieżnego, 141
  dynamiczne skalowanie poziome, 574
  filtr Transform, 227
  integracji, 600
    API proxy, 600
    broker komunikatów, 604
    koordynacja API, 601
  iteracja sekwencyjna z obietnicami, 167
  iteratora sekwencyjnego, 137
  nieograniczonego wykonywania współbieżnego, 142
  obserwatora, 108
  odwrócenie sterowania, IoC, 302
  ograniczonej współbieżności, 147
  proaktora, 38
  reaktora, 36
  rejestr usług, 575
  ujawnianego modułu, 49
  wykonywania sekwencyjnego, 133
  wywołań zwrotnych, 92

Z

zadania współbieżne, 142, 144
zasada
  DRY, 523
  KISS, 30
  wczesnego zwrotu, 130
zdalne
  rejestrowanie danych, 255
  wywoływanie procedur, RPC, 611
zdarzenia, 91
  asynchroniczne, 114
  demultipleksacja, 34
  nasłuchiwanie, 109, 113
  synchroniczne, 114
  śledzenie, 117
ZeroMQ, 626
  gniazda PUB i SUB, 627
  łamanie skrótów, 648
  oprogramowanie pośredniczące, 386
  wzorzec fan-in, 647
  wzorzec fan-out, 647
zmienna globalna
  globalThis, 81
  process, 42
  utils, 50

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

Mistrzowskie opanowanie Node.js — od podstaw po systemy produkcyjne

Node.js to obecnie jedna z najpopularniejszych platform do tworzenia aplikacji serwerowych, używana przez miliony programistów na całym świecie. Jej asynchroniczna, sterowana zdarzeniami architektura idealnie sprawdza się w budowaniu skalowalnych systemów internetowych, API i mikrousług. Książka odpowiada na potrzeby współczesnych programistów, prezentując najnowsze funkcje platformy, sprawdzone wzorce projektowe, a także praktyczne techniki tworzenia wydajnych, niezawodnych aplikacji produkcyjnych.

Autorzy prowadzą czytelnika przez wszystkie aspekty profesjonalnego programowania w Node.js — od fundamentów platformy i systemu modułów, przez programowanie asynchroniczne z wykorzystaniem callbacks, obietnic i async/await, aż po zaawansowane wzorce projektowe i architektury mikrousług. Szczegółowo omawiają strumienie Node.js, konstrukcyjne, strukturalne i behawioralne wzorce projektowe. Książka zawiera praktyczne receptury odpowiadające na typowe wyzwania: asynchroniczną inicjalizację, buforowanie, anulowanie operacji czy wykonywanie zadań obciążających procesor. Końcowe rozdziały koncentrują się na skalowalności — przedstawiają techniki skalowania aplikacji, zastosowanie Kubernetes i wzorce komunikacji w systemach rozproszonych z użyciem Redis, RabbitMQ i ZeroMQ.

Najważniejsze zagadnienia:

- Asynchroniczne wzorce kontroli przepływu
- Wzorce projektowe dostosowane do Node.js
- Testowanie aplikacji — testy jednostkowe, integracyjne i E2E
- Zaawansowane receptury
- Skalowanie i architektura
- Komunikacja w systemach rozproszonych

Luciano Mammino — senior architekt w firmie fourTheorem, AWS Serverless Hero i Microsoft MVP z ponad 15-letnim doświadczeniem. Aktywny prelegent, prowadzi blog *loige.co.*

Mario Casciaro — architekt oprogramowania i przedsiębiorca. Jego kod jest używany w operacjach ratowania astronautów.

HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-289-3647-8
9 788328 936478

Cena: 129,00 zł