DOI: https://doi.org/10.33216/1998-7927-2020-264-8-15-19

УДК 004.4'236

ОСОБЛИВОСТІ МАСШТАБУВАННЯ ВЕБ-ДОДАТКІВ

Щербаков Є. В., Щербакова М. Є.

SCALING FEATURES OF WEB APPLICATIONS

Shcherbakov E.V., Shcherbakova M.E.

Node.js - це відносно нова платформа для розробки веб-додатків, серверів додатків, будь-якого типу мережевих серверних або клієнтських модулів, а також розробки програм загального призначення. Вона спроектована з можливістю максимальної масштабованості мережевих додатків завдяки оригінальному поєднанню можливостей серверної мови JavaScript, асинхронного введення-виведення і асинхронного програмування. У той час, коли в багатьох платформах потоки широко використовуються для масштабування додатків з метою максимального завантаження CPU, Node.js уникає потоків через їх внутрішню складність і великі накладні витрати при безкінечних переключеннях між ними. Додатки на платформі Node.js зазвичай потребують масштабування раніше, ніж традиційні веб-сервери, щоб забезпечити можливість користування всіма ресурсами додатка. Це змушує розробників розглядати можливість масштабування на самих ранніх етапах розробки й пересвідчуватися, що додаток не покладається на ресурси, які не можуть спільно використовуватися декількома процесами або комп'ютерами. Екземпляри додатків не повинні зберігати інформацію на ресурсах, які не підтримують спільного використання, таких як пам'ять або диск, що фактично є обов'язковою умовою масштабування. Масштабування сприятливо діє на інші характеристики додатка, зокрема, на надійність і відмовостійкість, бо дозволяє гарантувати певний рівень обслуговування навіть при наявності неполадок і збоїв. Коли запускається кілька екземплярів одного і того ж додатка, формується надлишкова система, тобто, якщо один з екземплярів припинить роботу, інші продовжать обслуговування запитів. Цю схему легко реалізувати за допомогою модуля cluster в додатках Node.js. Альтернативою використання модуля cluster є запуск декількох автономних екземплярів одного і того ж додатка, які прослуховують різні порти або виконуються на різних комп'ютерах, і використання зворотного проксі-сервера для доступу до екземплярів і розподілу трафіку між ними. Ще одним варіантом є розподіл додатку на відносно автономні компоненти, які оформлені як окремі, незалежні додатки, що мають власні бази даних. В статті показано, що можна виконувати масштабування веб-додатків шляхом їх декомпозиції на підставі виконуваних ними функцій і сервісів. Ця методика дозволяє масштабувати не тільки пропускну здатність додатків, але, що головніше, їх складність.

Ключові слова: *JavaScript, веб-сервер, Node.js, масштабування додатків, розподілені системи*

Вступ. Характерні особливості платформи Node.js роблять її ідеальною для реалізації розподілених систем, які складаються з вузлів, взаємодіючих по мережі. Одного потоку виконання, який раціонально використовує неблокуюче введення/виведення, цілком достатньо для додатків, що обробляють помірне число запитів, зазвичай, кілька сотень в секунду (багато в чому залежить від особливостей додатка). Яким би потужним не було апаратне забезпечення, пропускна здатність одного потоку виконання все одно обмежена, тому якщо потрібно використовувати Node.js для створення високонавантажених додатків, виникає необхідність у масштабуванні платформи для використання в додатках кількох процесів і комп'ютерів [1].

Однак високе навантаження - не єдина причина масштабування додатків на платформі Node.js; фактично, використовуючи ті ж прийоми, можна поліпшити інші характеристики додатка, такі як надійність і стійкість до збоїв. Крім того, поняття «масштабованість» відноситься також до розміру і складності додатка, оскільки створення архітектури, здатної розширюватися, грає важливу роль при розробці програмного забезпечення. Тим більше, що використання JavaScript зобов'язує розробників зберігати додаток простим і розбивати його на керовані частини, що полегшує масштабування і розподіл навантаження [2].

Клонування і розподіл навантаження в Node.js. Традиційні багатопотокові веб-сервери зазвичай масштабуються, тільки коли неможливо наростити ресурси комп'ютера, або коли це є більш дорогою операцією, ніж підключення до роботи ще одного комп'ютера. Використовуючи механізм багатопотоковості, традиційні веб-сервери можуть задіяти всю обчислювальну потужність сервера, використовуючи всі доступні процесори і пам'ять.

Додатки на платформі Node.js зазвичай починають потребувати масштабування набагато раніше, ніж традиційні веб-сервери, навіть в одному ком-

п'ютері з тим, щоб мати можливість користуватися всіма його ресурсами [3].

Не варто вважати це недоліком. Навпаки, практично вимушене масштабування сприятливо діє на інші характеристики додатка, зокрема на надійність і відмовостійкість. Справді, масштабування додатка Node.js шляхом клонування є відносно простим і часто реалізується, навіть якщо немає необхідності задіяти більше ресурсів, а тільки з метою забезпечення надлишковості і стійкості до збоїв.

Модуль cluster. Базовий механізм для розподілу навантаження додатка між декількома екземплярами Node.js, запущеними на одному комп'ютері, заснований на модулі cluster, що входить до складу бібліотеки ядра. Модуль cluster спрощує розгалуження програми на кілька процесів і автоматично розподіляє між ними вхідні з'єднання, як це показано на схемі рис. 1.

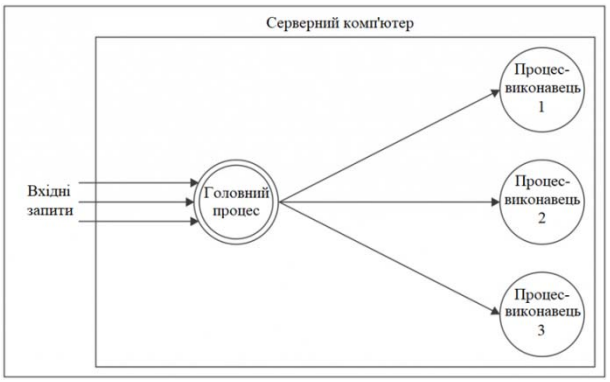

Рис. 1. Масштабування розгалуженням процесу

Головний процес (master process) відповідає за породження ряду процесів-виконавців (worker processes), кожен з яких представляє собою екземпляр масштабованого додатка. При цьому всі вхідні запити розподіляються між процесами-виконавцями, використовуючи циклічний алгоритм, який гарантує рівномірний розподіл навантаження між усіма процесами-виконавцями.

Створення простого HTTP-сервера. В якості прикладу створимо HTTP-сервер app.js, а потім здійснимо його клонування і розподіл навантаження за допомогою модуля cluster:

```
const http = require('http');
const pid = process.pid;
http.createServer((req, res) => {
// Імітація інтенсивної роботи процесора
let i = 1e7; while (i > 0) { i-- }
    console.log(`Обробка запиту ${pid}`);
    res.end(`Hello from ${pid}\n`);
}).listen(8080, () => {
    console.log(`Стартував ${pid}`);
});
```

У відповідь на будь-який запит HTTP-сервер буде повертати повідомлення з його ідентифікатором процесу (PID), що дозволяє визначити екземпляр додатку, який обробляє запит. Крім того, для імітації навантаження на центральний процесор передбачено виконання 10 мільйонів порожніх циклів, без чого сервер не буде мати практично ніякого навантаження, крім невеликих тестових видач.

Для того, щоб масштабувати додаток app.js за допомогою модуля cluster, створимо ще один модуль clusteredApp.js:

```
const cluster = require('cluster');
const os = require('os');
if(cluster.isMaster) {
    // 1
   const cpus = os.cpus().length;
   console.log(`Кластеризація на ${cpus} CPU`);
   for (let i = 0; i<cpus; i++) {
      cluster.fork();
   }
} else {
    // 2
   require('./app');
}
```

Як видно, використання модуля cluster не вимагає особливих зусиль. Пояснимо, що відбувається:

1. Запуск модуля clusteredApp з командного рядка фактично означає запуск головного процесу. Змінна cluster.isMaster автоматично отримує значення true, і вся робота зводиться до розгалуження процесу викликом методу cluster.fork(). У приведеному прикладі визначається кількість процесорів в системі і запускається така ж кількість робочих процесів, що дозволяє задіяти всі доступні обчислювальні потужності.

2. Виклик методу cluster.fork() в головному процесі здійснює повторний запуск все того ж головного модуля (clusteredApp), але на цей раз у вигляді процесу-виконавця (змінна cluster.isWorker отримує значення true, а змінна cluster.isMaster - значення false). Коли додаток запускається як процес-виконавець, він може почати виконувати якусь реальну роботу. В даному прикладі завантажується модуль app.js, що призводить до запуску нового HTTP-сервера.

Важливо не забувати, що кожен виконавець - це окремий процес Node.js з власним циклом подій, простором пам'яті і завантаженими модулями.

Внутрішньо модуль cluster використовує метод child_process.fork(), тому автоматично утворюються комунікаційні канали між головним процесом і процесами-виконавцями. Екземпляри процесів-виконавців доступні через змінну cluster.workers, тому для відправки всім їм повідомлення досить наступних рядків коду:

```
Object.keys(cluster.workers).forEach(id => {
cluster.workers[id].send('Привіт від головного процесу');
});
```

Масштабування за допомогою зворотного проксування. Використання модуля cluster - не єдиний спосіб масштабування веб-додатків на платформі Node.js. Фактично традиційні методи часто виявляються кращими, оскільки забезпечують більш повний контроль і ширші можливості в середовищах з високою доступністю.

Альтернативою використанню модуля cluster є запуск декількох автономних екземплярів одного і того ж додатка, які прослуховують різні порти або виконуються на різних комп'ютерах, і використання зворотного проксі-сервера (або шлюзу) для доступу до екземплярів і розподілу трафіку між ними [4]. У такій конфігурації відсутній головний процес, який розподіляє запити між процесами-виконавцями, натомість є багато автономних процесів, запущених на виконання на одному комп'ютері (таких, що прослуховують різні порти) або розкиданих по декількох комп'ютерах в мережі. Роль єдиної точки доступу до додатка грає зворотний проксі-сервер - спеціальний модуль або сервіс, що знаходиться між клієнтами і екземплярами додатка, який приймає всі запити, передає їх кінцевому серверу і повертає результат клієнтові, ніби він сам його створив. У цьому випадку зворотний проксі-сервер використовується також в якості балансувальника навантаження, розподіляючи запити між екземплярами додатка.

На рис. 2 зображена схема типової конфігурації зворотного проксі-сервера, який виступає в ролі балансувальника навантаження між кількома процесами, запущеними на кількох комп'ютерах.

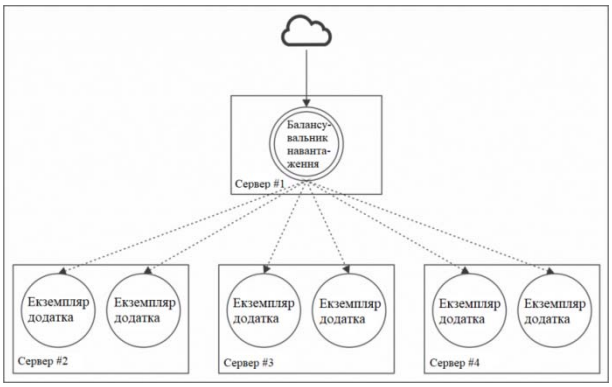

Рис. 2. Типова конфігурація зворотного проксі-сервера

Декомпозиція складних додатків. Можна виконати масштабування додатка шляхом його декомпозиції на підставі виконуваних ним функцій і сервісів. Ця методика дозволяє масштабувати не тільки пропускну здатність додатка, але, що головніше, його складність.

Монолітна архітектура. Монолітна архітектура програмного комплексу не завжди означає систему без модулів, в якій всі служби взаємопов'язані і практично нероздільні. Часто монолітні системи мають модульну архітектуру і гарний розподіл внутрішніх компонентів.

Хорошим прикладом є ядро Linux, яке відноситься до категорії так званих монолітних ядер (що суперечить принципам екосистеми і філософії Unix). Ядро Linux містить тисячі сервісів і модулів, які можна завантажувати і вивантажувати динамічно, навіть під час роботи системи. Однак всі вони виконуються в режимі ядра, що може приводити до неминучого краху операційної системи при виникненні збою в будь-якому з них.

Альтернативою монолітній архітектурі є мікроядерна архітектура, в якій тільки основні сервіси операційної системи працюють в режимі ядра, а всі інші виконуються в режимі користувача і, як правило, кожен в своєму власному процесі. Основна перевага такого підходу в тому, що збій в будь-якому з цих сервісів зазвичай призводить тільки до краху цього сервісу, не впливаючи на стабільність всієї системи.

Сучасні монолітні додатки можна порівняти з монолітними ядрами ОС. Збій в будь-якому компоненті впливає на всю систему, тобто, якщо перевести в терміни платформи Node.js, це означає, що всі сервіси є частиною однієї кодової бази і виконуються в одному процесі (якщо не клонуються).

На рис. 3 зображений приклад монолітної архітектури веб-додатка.

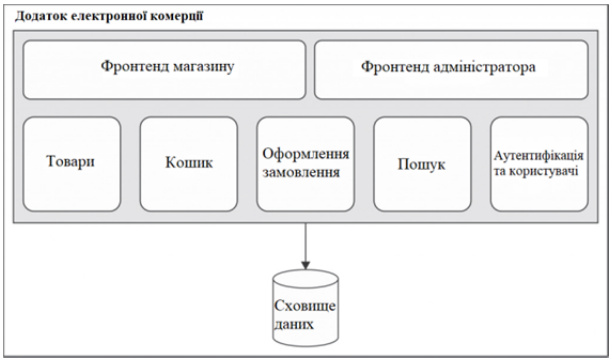

Рис. 3. Приклад монолітної архітектури типового веб-додатка

Представлений додаток має модульну структуру. У ньому передбачені два різні інтерфейси: один - основний - для магазину і другий - для адміністрування. Є також чіткий поділ сервісів, кожен з яких відповідає за конкретну частину бізнес-логіки: товари, кошик, оформлення замовлення, пошук і аутентифікацію користувачів. Однак його архітектура є монолітною, оскільки всі модулі, по суті, є частиною однієї і тієї ж кодової бази і запускаються як частина одного додатка. Збій в будь-якому з компонентів, наприклад, не перехоплений виняток, може привести до краху всього інтернет-магазину.

Мікросервісна архітектура. Ідея полягає в тому, щоб розділити додаток на відносно автономні компоненти, оформивши їх як окремі, незалежні додатки. Це повна протилежність концепції монолітної архітектури. На рис. 4 зображена схема додатка електронної комерції з мікросервісною архітектурою.

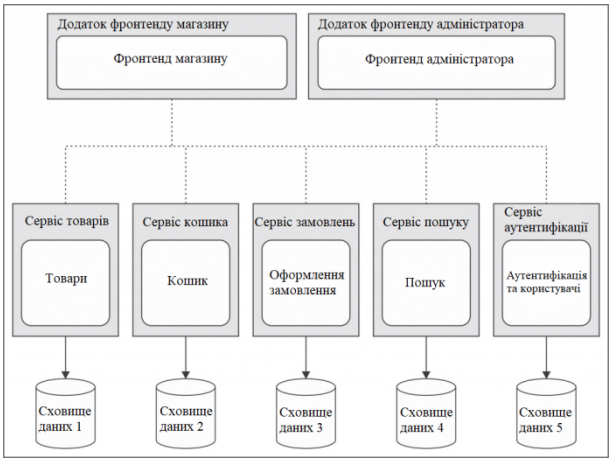

Рис. 4. Приклад мікросервісної архітектури веб-додатка

Кожен з основних компонентів електронної комерції тепер є самостійною незалежною сутністю, знаходиться в окремому контексті і має власну базу даних. Всі компоненти є незалежними додатками, які надають групу пов'язаних між собою сервісів (висока зчепленість).

Володіння сервісом власними даними є важливою характеристикою мікросервісної архітектури. Ось чому база даних також повинна розділятись для підтримки належного рівня ізоляції та незалежності. Якщо використовується загальна база даних, сервісам стане набагато легше працювати разом; однак це також призводить до зв'язності між сервісами (загальні дані), зводячи нанівець деякі переваги існування різних додатків.

Висновок. Поділ великого монолітного додатка на ряд невеликих сервісів дозволяє створювати незалежні одиниці, які без особливих зусиль можуть повторно використовуватися.

Головною перевагою мікросервісів є те, що рівень приховування інформації в них, як правило, набагато вище в порівнянні з монолітним додатком. Це стає можливим через те, що взаємодія зазвичай відбувається через віддалений інтерфейс, наприклад, через веб-сервіс або брокер повідомлень, що робить набагато простішим приховування деталей реалізації і захист клієнта від впливу змін в реалізації або розгортанні сервісу. Наприклад, щоб викликати веб-сервіс, не потрібно знати особливості масштабування його інфраструктури, якою мовою програмування він написаний, яка база даних використовується і таке інше.

Література

1. Herron D. Node.js Web Development. Packt Publishing Ltd.: Birmingham, UK, 2020. 765 p.
2. Casciaro M., Mammino L. Node.js Design Patterns, Third Edition. Packt Publishing Ltd.: Birmingham, UK, 2020. 661 p.
3. Node.js v15.1.0 Documentation [Електронний ресурс] URL: https://nodejs.org/dist/latest-v15.x/docs/api/
4. Thomas Hunter II. Distributed Systems with Node.js. O'Reilly Media: Sebastopol, California, 2021. 557 p.

References

1. Herron D. Node.js Web Development. Packt Publishing Ltd.: Birmingham, UK, 2020. 765 p.
2. Casciaro M., Mammino L. Node.js Design Patterns, Third Edition. Packt Publishing Ltd.: Birmingham, UK, 2020. 661 p.
3. Node.js v15.1.0 Documentation [Electronic resource] URL: https://nodejs.org/dist/latest-v15.x/docs/api/
4. Thomas Hunter II. Distributed Systems with Node.js. O'Reilly Media: Sebastopol, California, 2021. 557 p.

Щербаков Е.В., Щербакова М.Е. Особенности масштабирования веб-приложений

Node.js - это относительно новая платформа для разработки веб-приложений, серверов приложений, любого рода сетевых серверных или клиентских модулей, а также разработки программ общего назначения. Она спроектирована с возможностью максимальной масштабируемости сетевых приложений благодаря оригинальному сочетанию возможностей серверного языка JavaScript, асинхронного ввода-вывода и асинхронного программирования. В то время, когда во многих платформах потоки широко используются для масштабирования приложений с целью максимальной загрузки CPU, Node.js избегает потоков из-за их внутренней сложности и больших накладных расходов при бесконечных переключениях между ними. Приложения на платформе Node.js обычно требуют масштабирования ранее, чем традиционные веб-серверы, чтобы обеспечить возможность использования всех ресурсов приложения. Это вынуждает разработчиков рассматривать возможность масштабирования на самых ранних этапах разработки и удостоверяться, что приложение не полагается на ресурсы, которые не могут совместно использоваться несколькими процессами или компьютерами. Экземпляры приложений не должны хранить информацию на ресурсах, которые не поддерживают совместное использование, таких как память или диск, что фактически является обязательным условием масштабирования. Масштабирование благоприятно воздействует на другие характеристики приложения, в частности, надежность и отказоустойчивость, так как позволяет гарантировать определенный уровень обслуживания даже при наличии неполадок и сбоев. Когда запускается несколько экземпляров одного и того же приложения, формируется избыточная система, то есть, если один из экземпляров прекратит работу, другие продолжат обслуживание запросов. Эту схему легко реализовать с помощью модуля cluster в приложениях Node.js. Альтернативой использования модуля cluster является запуск нескольких автономных экземпляров одного и того же приложения, которые прослушивают различные порты или выполняются на разных компьютерах, и использование обратного прокси-сервера для доступа к экземплярам и распределения трафика между ними. Еще одним вариантом является распределение приложения на относительно автономные компоненты, которые оформлены как отдельные, независимые приложения, имеющие собственные базы данных. В статье показано, что можно выполнять масштабирование веб-приложений путем их декомпозиции на основании выполняемых ими функций и сервисов. Эта методика позволяет масштабировать не только пропускную способность приложений, но, что главнее, их сложность.

Ключевые слова: *JavaScript, веб-сервер, Node.js, масштабирование приложений, распределенные системы*

Shcherbakov E.V., Shcherbakova M.E. Scaling features of web applications

Node.js is a relatively new platform for developing web applications, application servers, any type of network server or client modules, and developing general-purpose applications. It is designed to maximize the scalability of network applications thanks to the original combination of JavaScript server language capabilities, asynchronous I/O and asynchronous programming. While on many platforms threads are widely used to scale applications to maximize CPU usage, Node.js avoids threads because of their intrinsic complexity, as well as the high overhead of endless switching between them. Applications on the Node.js platform typically need to be scaled earlier than traditional web servers to ensure all application resources can be used. This forces developers to consider scaling at the earliest stages of development and make sure the application does not rely on resources that cannot be shared by multiple processes or computers. Application instances should not store information on resources that do not support sharing, such as memory or disk, which is actually a prerequisite for scaling. Scaling has a positive effect on other application characteristics, in particular, on reliability and fault tolerance, because it guarantees a certain level of service even when malfunctions and failures happens. When multiple instances of the same application are launched, a redundant system is formed, so if one instance stops working, the others will continue to service requests. This scheme is easy to implement using the cluster module in Node.js applications. An alternative to using the cluster module is to run multiple stand-alone instances of the same application that listen on different ports or run on different computers, and use a reverse proxy server to access instances and distribute traffic between them. Another option is to divide the application into relatively stand-alone components, which are designed as separate, independent applications with their own databases. The article shows possibility to scale web applications by decomposing them on the basis of their functions and ser-vices. This technique allows to scale not only the bandwidth of applications, but most importantly, their complexity.

Keywords: *JavaScript, web server, Node.js, application scaling, distributed systems*

Щербаков Є.В. – к.т.н., доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: gkvarc@gmail.com

Щербакова М. Є. – к.т.н., доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: m.shcherbakova432@gmail.com

Стаття подана 10.11.2020.