УДК 004.4
DOI: 10.18523/2617-3808.2024.7.31-39

Колінько П. В.

АВТОМАТИЗОВАНА ГЕНЕРАЦІЯ І НАЛАШТУВАННЯ МІКРОСЕРВІСІВ ДЛЯ СПРОЩЕННЯ ПРОЦЕСУ РОЗРОБЛЕННЯ

У статті розглянуто підходи і методи автоматизованої генерації коду та структури застосунків, зокрема таких, що базуються на мікросервісній архітектурі. Описано розроблений програмний застосунок для генерації мікросервісної архітектури засобами платформи Node.js. В основу розробки покладено використання автоматизованої генерації програмного коду та архітектури на базі поняття scaffolding.

Ключові слова: scaffolding, мікросервісна архітектура, генерація коду.

Вступ

Розроблення вебзастосунків завжди починається з аналізу бізнес-вимог і підбору архітектури. Одним із важливих питань, що постає на початку розроблення бекенд-застосунків, є вибір поміж монолітною і мікросервісною архітектурами, кожна з яких має свої переваги та недоліки. Проте варто зазначити, що у кожної із зазначених архітектур є спільний етап конфігурації, що передбачає налаштування середовища, встановлення бібліотек, підключення бази даних і багато супутніх кроків. У разі розроблення мікросервісного застосунку, процес конфігурації повторюється велику кількість разів, адже кожен сервіс є окремим застосунком в ізольованому середовищі, що значно сповільнює процес імплементації.

Для вирішення проблеми повільного розроблення і запобігання помилкам у момент конфігурації використовують scaffolding-утиліти. Scaffolding передбачає певну автоматизацію генерації коду, шаблону, архітектури проєкту. На базі платформи Node.js існують такі генеративні утиліти, як Nest CLI, Yeoman, Prisma CLI тощо. Однак жодна з перелічених бібліотек не вирішує проблеми автоматизованої генерації мікросервісів.

Монолітна архітектура — це класичний підхід у розробленні програмного забезпечення, особливо для бекенд-застосунків, який передбачає, що проєкт є цілісною та незалежною сутністю. На противагу, мікросервісна архітектура передбачає, що найбільшою незалежною одиницею розподіленої системи є один мікросервіс, що представлений у формі атомарного застосунку. Зазвичай кожен такий сервіс побудований довкола конкретної бізнес-логіки, наприклад, окремий сервіс для аутентифікації, листування тощо. Кожен з них є розгорнутим в окремому середовищі, зазвичай має власну базу даних.

Структура проєкту, основаного на мікросервісній архітектурі

Через особливості своєї структури розподілені системи мають додатковий функціонал для правильної взаємодії між собою, оскільки під час розроблення мікросервісів легко втратити контроль над їхньою кількістю. Завершений мікросервісний додаток має структуру, представлену на рис. 1.

Найвищим рівнем завершеного доробку є клієнт. Зазвичай фронтенд-застосунок, що має графічний інтерфейс, зрозумілий людині, з якою взаємодіє користувач. Комунікація між клієнтом і сервером у монолітній і мікросервісній архітектурах не відрізняється, тобто запити надсилають за допомогою API.

Наступним рівнем є функціональна бізнес-логіка, що відповідає за оброблення інформації, збір статистики, впровадження відкладених задач тощо. Його поділяють на два підтипи:

- experience;
- domain.

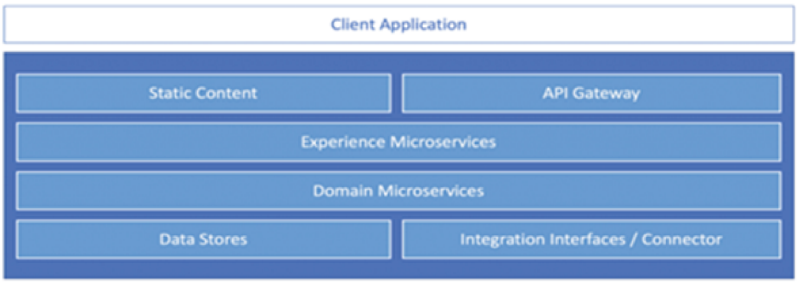

Рис. 1. Структура мікросервісного застосунку

Рівень experience відповідає за бізнес-логіку і взаємодію з даними, клієнтом та іншими навантаженими частинами застосунку, натомість domain відповідає за вузькоспеціалізований функціонал, як-от шифрування даних.

Використання окремого середовища для сервісів передбачає необхідність комунікації між ними. Обмін даними та повідомленнями між сервісами відбувається за допомогою Event Brokers, Message Queues або інших патернів взаємодії сервісів. Такі додаткові утиліти дають змогу асинхронно передавати інформацію від сервісу до сервісу. Черги повідомлень, такі як RabbitMQ, Apache Kafka, — це зовнішні утиліти, що дають змогу асинхронно обмінюватись даними між різними компонентами застосунку безпосередньо. Окрім мікросервісної архітектури, можуть бути використані в рамках моноліту чи хмарної архітектури. Вони працюють за принципом обміну повідомленнями від одного ресурсу (який називають publisher або producer) до іншого (consumer). Оскільки операція є асинхронною, тобто такою, що час для її виконання є невідомим, повідомлення публікуються у так звану чергу, де зберігаються за принципом «отримати і забути». Брокери подій (event brokers) є схожим механізмом до черги повідомлень, тобто це такі посередники, що обробляють дані в момент їх передавання від publisher до subscriber. Вони мають доступ до контексту та метаданих самого повідомлення й можуть робити з ними такий набір операцій, як читання та запис. Проте, на відміну від черг повідомлень, брокери подій реалізовані так, що publisher і subscriber не обов'язково мають бути напряму пов'язані між собою, адже за керування «напрямком» інформації відповідає саме такий посередник, що надсилає дані від ресурсу-публікатора повідомлення до одного або декількох підписників.

Комунікація клієнта і сервера в рамках мікросервісної архітектури відбувається або з кожним сервісом окремо, або за допомогою шлюзу API Gateway. Шлюз — це єдина точка входу в додаток. Фактично, це окремий застосунок, який розсилає запити на дочірні сервіси всього застосунку і повертає відповідь клієнту.

Кожен сервіс у рамках мікросервісної архітектури є незалежним додатком, тому він має бути розгорнутим в окремому середовищі. Для спрощення цього процесу використовують контейнеризацію, яка передбачає розгортання програмного застосунку в ізольованому віртуальному середовищі, умовному «контейнері», що не взаємодіє із зовнішніми компонентами системи, проте має змогу звертатися до них. Контейнеризація реалізується найчастіше за допомоги спеціальних програмних продуктів типу Docker або Kubernetes. Останнє програмне забезпечення є оркестратором, тобто «центром керування» всіх контейнеризованих сервісів, що спрощує і пришвидшує процес інтеграції, розроблення та паралельного розгортання декількох версій продукту.

Scaffolding у програмуванні

Підхід scaffolding передбачає генерацію за якимось загальним шаблоном або «скелетом». Ідеться необов'язково про генерацію кінцевої версії продукту, що є готовим до використання, навпаки, частіше згенерована частина є лише основою, що спрощує та значно пришвидшує подальше розроблення застосунку.

Scaffolding можна поділити на дві основні категорії: генерацію коду і генерацію архітектури.

У першому випадку утиліта допомагає генерувати шматки типового коду, такі як інтерфейси, функції getter і setter тощо. Цей підхід підтримується деякими model-view-controller (MVC) фреймворками.

Типовий приклад генерації коду — scaffolding у RoR (Ruby on Rails), що працює за допомогою CLI (comand line interrface) і передбачає набір консольних команд, які виконують будь-який функціонал у рамках проєкту, зазвичай різноманітні програмні скрипти.

На противагу генерації в рамках окремого фреймворку, така утиліта, як Yeoman, має набагато ширший спектр використання, наприклад генерацію фронтед- і бекенд-застосунків на різних мовах програмування. Проте наразі не створено такого yeoman-генератора, що вирішував би задачу генерації мікросервісної архітектури та окремих сервісів. Розробити такий генератор цілком можливо, проте такий підхід має декілька недоліків. По-перше, необхідно повністю підв'язуватись під можливості Yeoman, тобто в разі припинення підтримки коду генератор буде так само важко підтримувати. По-друге, можливості SDK є обмеженими і нерідко застарілими, тоді як Node.js має ширший і новіший функціонал.

Отже, розроблення застосунку для генерації мікросервісів, що є глобально незалежним від сторонніх бібліотек, є більш оптимальним і стабільним рішенням, адже у цьому випадку єдина технологія, що потрібна для «життя» проєкту, — це платформа Node.js. Ця платформа з часом усе більше і більше набирає популярності, що означає її постійну підтримку з боку розробників. Тому кінцевим варіантом застосунку для генерації мікросервісів стало поєднання найкращих практик із кожної дослідженої утиліти — простота у використанні за допомогою CLI команд і узагальнений широкий підхід, оснований на філософії yeoman-generators.

Архітектура програмного застосунку

Перед початком розроблення власного scaffolding-застосунку для генерації мікросервісів необхідно визначитися з архітектурою. Потрібно створити максимально зрозумілий користувацький інтерфейс за допомогою CLI, в якому не буде потреби довго розбиратись. Тому з метою уникнення потреби проходити «опитування», як у випадку роботи застосунку Yeoman, було вирішено описувати всі необхідні конфігурації в одному джерелі — конфігураційному файлі. Такий опис конфігурацій запозичено з підходу Iac (Infrastructure as a Code, інфраструктура як код) для опису та керування інфраструктурою хмарних сервісів за допомогою конфігураційних файлів. Первинний вигляд архітектури програмного застосунку та структуру згенерованого проєкту схематично зображено на рис. 2 і 3.

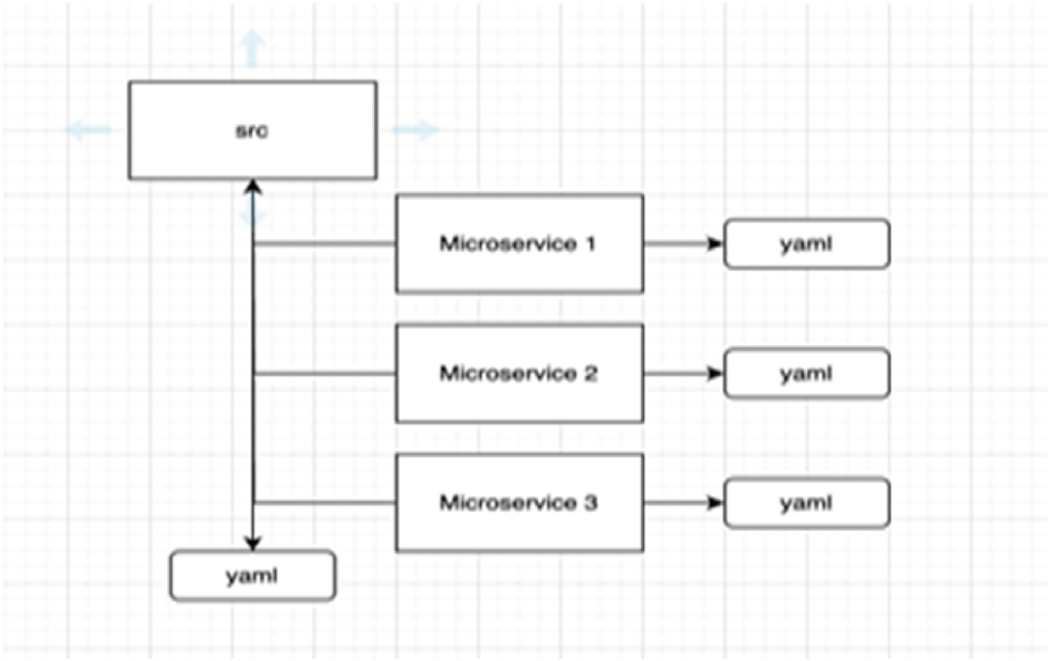

Рис. 2. Первинна архітектура проєкту

Проте під час роботи над застосунком багато початкових рішень було замінено на більш оптимальні. Наприклад, було прийнято рішення про спільний файл, що відповідатиме за налаштування проєкту й кожного сервісу. У такому разі весь додаток можна буде контролювати з «одного джерела» і користувачеві не доведеться робити зайву роботу, як-от створювати різні папки та файли.

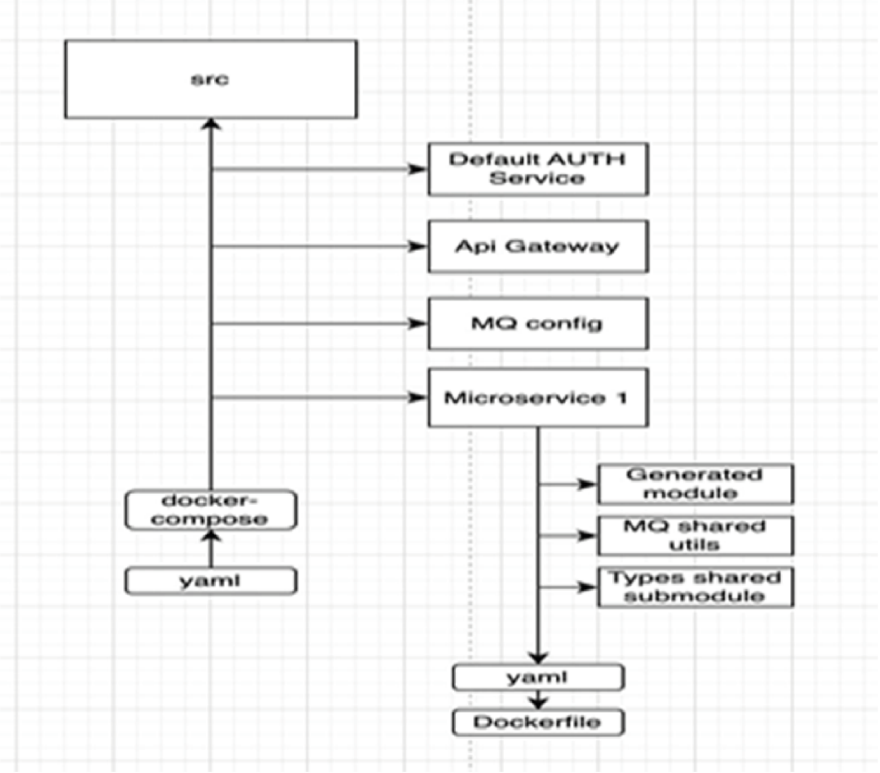

Рис. 3. Архітектура проєкту після виконаної генерації

Друга важлива зміна, яку було впроваджено, — це зміна формату конфігураційного файлу .yaml на формат .json, адже він є ідеально сумісним із мовою програмування Javascript.

Рис. 4. Приклад конфігураційного файлу microgen.json

Для старту генерації необхідно запустити команду *npm run generate - - -path=/path-to-microgen.json*. Консольна змінна -path вказує на абсолютний шлях до кореневого конфігураційного файлу.

Опис використаних технологій

Як мову програмування було вибрано Typescript. Розроблення будь-якого програмного застосунку за допомогою Typescript має низку переваг і недоліків. З-поміж недоліків — потреба витрачати більше часу на розроблення безпосередньо. Проте для такого застосунку, як microgen, критично важливим є правильний, односторонній потік даних і «виразний» опис кожної окремої сутності проєкту. Такий вибір мови програмування дає змогу підтримувати наявний код, розширювати наявний функціонал, мінімізує кількість помилок, пов'язаних із неоднорідністю кодової бази.

Для старту генерації необхідно звернути увагу на консольні аргументи, що передаються під час ініціалізації, а саме змінну *path*. Оскільки скрипт має бути універсальним і дозволяти генерацію для будь-якої файлової системи, без прив'язки до конкретної директорії, необхідно дати змогу користувачеві динамічно зазначати абсолютне розташування microge.json файлу. І для того, щоб отримати такий шлях до файлу, потрібно передати консольні аргументи і розпарсити їх. Для оброблення такого випадку вирішили відмовитися від сторонніх бібліотек і запровадити власне рішення, адже всі наявні варіанти не працюють належно.

Для генерації файлів, копіювання, зчитування, встановлення залежностей та інших операцій необхідно було обрати модулі їх виконання. В платформі Node.js існують дві вбудовані бібліотеки, які добре підійдуть для виконання поставленої задачі, — fs і node:child_process.

Для процесу логування даних вибрали зовнішню бібліотеку Winston, що імплементує головні методи виводу даних у консоль, відповідно до патерну Logger, такі як *log*, *info*, *warn* та *error*.

Модулі програмної реалізації

У цьому розділі описано, які рішення було впроваджено під час розроблення програмного забезпечення та аргументовано причину їх використання. Роботу такого застосунку організовано за принципом переходу від найменшої задачі до найбільшої. Такий підхід є необхідним, адже кожний наступний крок у генерації є фактично надбудовою над попереднім. Для спрощення розуміння та читабельності коду процес генерації було розбито на атомарні модулі, кожен з яких відповідає виключно за конкретну одиницю функціоналу.

Перший етап передбачає парсинг і валідацію головного конфігураційного файлу *microgen.json*. Читання файлу відбувається засобами бібліотеки fs:promises і записом даних у RAM. Оскільки формат файлу є .json, немає потреби в додаткових кроках, адже такий формат добре підходить для оброблення мовою Javascript.

```
export class ChildProcessBuilder {
  private query: string = "";
  private readonly executionHandler = null;
  constructor(customHandler?: any) {
    this.executionHandler = customHandler;
  }

  append(command: string): typeof this {
    if (!command) {
      return this;
    }
    if (!this.query) {
      this.query = command;
    } else {
      this.query = `${this.query} && ${command}`;
    }
    return this;
  }

  exec(): void {
    logger.info(`Executing query: ${this.query}`);
    nodeExec(this.query, (error, stdout) => {
      stdout && logger.info(stdout);
      error && logger.error(error);
    });
  }

  async execAsync(): Promise<any> {
    return util.promisify(require("node:child_process").exec)(this.query);
  }
}
```

Рис. 5. Імплементація класу ChildProcessBuilder

Валідація файлу впроваджується окремою функцією і є простою перевіркою на відповідність даних заданому шаблону: чи заповнені всі обов'язкові поля в загальній конфігурації проєкту або окремого мікрсервісу, чи існує імплементація вибраного варіанта. У разі невідповідності даних такому шаблону користувачу буде виведено повідомлення про помилку, де буде описано, що необхідно змінити для продовження роботи.

Важливо розглянути процес генерації коду засобами Node.js безпосередньо. Генерація коду відбувається в рамках атомарного модуля, що представлено як окрему функцію генерації. Для виконання консольних команд розроблено клас ChildProcessBuilder, що є імплементацію патерну Builder. Це такий патерн, що породжує шаблон об'єкта, або, як у випадку генерації коду, шаблон консольної команди, що потім буде виконана за допомогою метода exec.

За допомогою методу *append* клас об'єднує консольні команди в одну та зберігає їх у формі рядка. Цей клас має на меті виконання ланцюга команд послідовно. Оскільки процес, створений методом exec, є ізольованим, такий підхід — єдина можливість виконати набір команд послідовно.

Наступна проблема, яка виникла під час розроблення застосунку, — асинхронність виконання команд. Команда exec працює так, що час на виконання команди є невизначеним, проте виклик цього самого методу виконується синхронно. Для вирішення цієї проблеми було використано пакет utils.promisify, який працює подібно до fs:promises і дає змогу обробляти проміси за допомогою async/await синтаксису, що дає можливість очікувати нового завершення виконання консольної команди. Нижче наведено приклад створення команд для кодогенерації за допомогою ChildProcessBuilder.

```
export const initializeNodeProjects = (): Promise<any>[] => {
  const promises = __PROJECT_METADATA__.microservices
    .filter(({ framework }) => framework !== Frameworks.NEST)
    .map(({ absolutePath }) => {
      return new ChildProcessBuilder()
        .append(osExecutableCommands.changeDirectory(absolutePath))
        .append(npmExecutableCommands.npmInit())
        .execAsync();
    });

  logger.info("Initializing Node projects");

  return promises;
};
```

Рис. 6. Генерація директорій та ініціалізація порожнього Node.js проєкту

Список консольних команд було описано у формі пари ключ–значення, де ключ — назва команди, а значення — функція, яка повертає текстовий рядок із сигнатурою команди. Вони розділені на два типи — npm-команди та команди os. Список npm-команд відповідає за весь менеджмент, що пов'язаний із Node.js та npm, тоді як список os відповідає за команди операційної системи, наприклад створення директорії, копіювання файлів тощо. Кожна з цих команд описана у формі функції, а не у формі рядка, тому що деяким командам для роботи необхідно динамічно приймати аргументи. Завдяки такому підходу та можливостям мови програмування Typescript є змога уникнути помилки у разі, якщо у функцію буде передано аргумент, що не буде використаним. Тож можливо зіставити варіант із конфігураційного файлу і відповідну директиву, наприклад ініціалізацію фреймворку або мови програмування.

```
const frameworkConfigs: Record<string, CallableFunction> = {
  [Frameworks.EXPRESS]: npmExecutableCommands.npmInitExpress,
  [Frameworks.NEST]: npmExecutableCommands.npmInitNest,
  [Languages.TS]: npmExecutableCommands.npmInitTypeScript,
  [Languages.JS]: npmExecutableCommands.npmNoop,
  [Languages.DEFAULT]: npmExecutableCommands.npmNoop,
};      Pasha, last month • Added validation and nest generatic
```

Рис. 7. Приклад словника npm-команд для ініціалізації фреймворку

Оскільки проєкт написано мовою програмування Typescript, який своєю чергою транспілюється, тобто перекладається в Javascript, усі файли, що мали розширення не .js або .ts, не потрапляли в кінцевий Javascipt bundle, тобто побудований проєкт, готовий до використання. Такий підхід є стан-

дартною поведінкою tsc білдера, за допомогою якого Typescript перетворюється на Javascript, тому необхідно було розробити підхід, щоби всі ресурси проєкту зберігались у кінцевому продукті. Для цього розробили скрипт, що на етапі білд-проєкту дає змогу копіювати всю директорію із Typescript директорії в аналогічну директорію Javascript білда.

Вибудувана архітектура проєкту має чималу кількість переваг, як-от стабільність і безвідмовність роботи, універсальність, а головне — розширюваність, адже у розроблену кодову базу дуже легко додати нові фреймворки та конфігурації.

Аналіз розробленого програмного забезпечення

У результаті розроблення програмного забезпечення було досягнено основної мети зі створення стабільного застосунку, що є мінімально залежним від сторонніх бібліотек і має простий у використанні інтерфейс.

Єдина «точка входу», що представлена конфігураційним файлом, дає змогу не тільки полегшити процес розроблення на початкових етапах, а й легко керувати розширенням продукту, що використовує утиліту microgen. Наприклад, першочергові бізнес-умови проєкту вимагали створення лише двох сервісів, під назвою user і bookings.

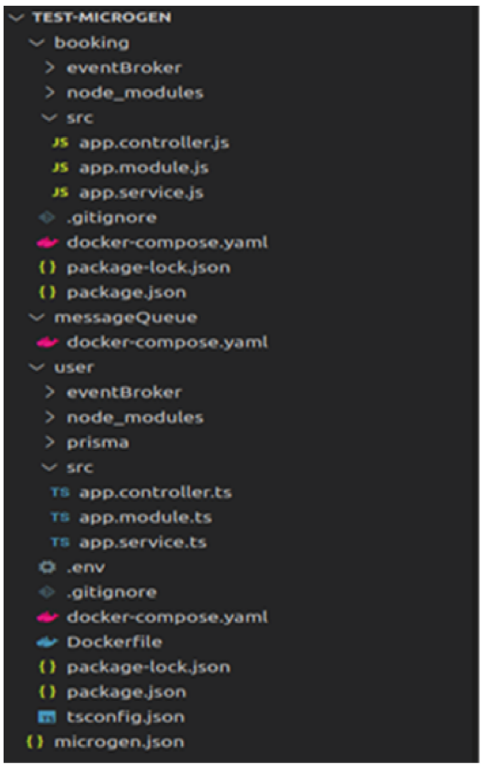

Рис. 8. Згенерована кодова база проєкту

Окрім мови програмування, в конфігураціях сервісів було вибрано фреймворк Express, необхідні node_modules пакети, які треба встановити на етапі генерації, бази даних postgresql та mysql із додатковими ORM prisma та typeorm відповідно. Для мікросервісу під назвою user також був вибрано параметр dockerFile, що відповідає за створення контейнера для згенерованого сервісу. Як показано на рис. 6, бажані сервіси було згенеровано з дотриманням конфігураційних вимог. Також було також додано сервіс для взаємодії із чергою повідомлень Apache kafka, зазначений у загальній конфігурації проєкту.

Якщо бізнес-вимоги до проєкту буде розширено, може постати потреба створити новий мікросервіс, наприклад засобами фреймворку Nest.js. Якщо додати в конфігураційний microgen.json файл опис такого сервісу та повторно використати CLI команду старту генерації, сервіс буде додано, а всі інші, що вже були згенеровані, не матимуть жодних змін, адже всі сервіси, що містять package.json файл, будуть проігноровані для повторної генерації. Такий підхід є необхідним для запобігання програмного втручання в зміни, внесені розробниками.

Висновок

Розроблений додаток microgen має низку вагомих переваг, таких як незалежність від бібліотек, що є нестабільними у підтримці, але і використовує вже розроблені рішення, такі як Nest CLI або Prisma CLI. Такий підхід дає змогу позбутись зайвої розробки додаткового коду та ресурсів застосунку, згенерувавши скелет мікросервісу, що використовує фреймворк Nest.js або ж ініціалізацію конфігураційного Prisma файлу, за допомогою однієї команди, без необхідності власноруч прописувати код. Гнучка архітектура застосунку microgen дозволяє швидко додавати нові ресурси та розширювати функціонал.

Проте розроблений додаток також має певні недоліки. Наприклад, наразі функціонал додатку є обмеженим певним набором технологій, тобто він здатен працювати лише з двома мовами програмування, двома фреймворками — Express і Nest.

Ще одним великим недоліком є брак функціоналу збереження версіонування (попереднього стану) проєкту. Підхід Iac та утиліта terraform, з яких було запозичено ідею єдиного конфігураційного файлу як основи генерації проєкту, мають функціонал, що дає змогу порівнювати попередню версію застосунку із теперішньою конфігурацією на наступному етапі генерації. Наприклад, такий функціонал є корисним для видалення конфігурацій, таких як зміна бази даних, відмова від використання docker-compose.yaml тощо.

Підсумовуючи, варто зазначити, що розроблена утиліта не має чітких відповідників у рамках поставленої задачі. Вона спрощує та пришвидшує розроблення мікросервісів на базі платформи Node.js, що можуть бути згенеровані на базі найпопулярніших технологій цієї платформи. Проте можливості застосунку обмежуються використанням невеликого переліку найпопулярніших технологій і відсутністю версіонування.

Список використаних джерел

1. Altcademy Team. What is Scaffolding in Ruby on Rails? [Electronic resource] // Altcademy Blog. — Mode of access: https://www.altcademy.com/blog/what-is-scaffolding-in-ruby-on-rails/#:~:text=Scaffolding%20in%20Ruby%20on%20Rails%20is%20a%20helpful%20feature%20that,practices,%20and%20speeds%20up%20development, 2024. — Title from screen.
2. Global Logic. Mikroservisna arkhitektura dlia pochatkivtsiv. Chastyna I [Electronic resource] // GlobalLogic Ukraine. — Mode of access: https://www.globallogic.com/ua/insights/blogs/microservices-architecture-for-beginners-part-one/. — Title from screen.
3. Hoang C. Monolith Architecture [Electronic resource] / C. Hoang // Medium. — Mode of access: https://tech.tamara.co/monolith-architecture-5f00270f384e. — Title from screen.
4. IBM. What Is a Message Queue? [Electronic resource] // IBM in Deutschland, Österreich und der Schweiz. — Mode of access: https://www.ibm.com/topics/message-queues. — Title from screen.
5. Microsoft. What is infrastructure as code (IaC)? Azure DevOps. Microsoft Learn: Build skills that open doors in your career [Electronic resource]. — Mode of access: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code. — Title from screen.
6. Nagpal A. Monolithic vs Microservices Architecture: Advantages, Disadvantages, And Differences [Electronic resource] / A. Nagpal // Medium. — Mode of access: https://medium.com/@jasminepuno/monolithic-vs-microservices-architecture-advantages-disadvantages-and-differences-2bee6d1da8ca#:~:text=Monolithic%20architecture%20is%20a%20conventional,closely%20connected%20and%20centralized%20system. — Title from screen.
7. Naik A. How to scaffold ExpressJS server and test it [Electronic resource] / A. Naik // Medium. — Mode of access: https://medium.com/craft-academy/how-to-scaffold-expressjs-server-and-test-it-d2a2ab1d30e0. — Title from screen.
8. NestJS. Documentation | NestJS — A progressive Node.js framework [Electronic resource]. — Mode of access: https://docs.nestjs.com/. — Title from screen.
9. NodeJS. Child process | Node.js v22.2.0 Documentation. Node.js – Run JavaScript Everywhere [Electronic resource]. — Mode of access: https://nodejs.org/api/child_process.html. — Title from screen.
10. Solace. What is an Event Broker? [Electronic resource] // Solace. — Mode of access: https://solace.com/what-is-an-event-broker/. — Title from screen.
11. Yeoman. Getting started with Yeoman [Electronic resource] // Yeoman. — Mode of access: https://yeoman.io/learning/. — Title from screen.

References

Documentation: Nestjs — a progressive node.js framework. NestJS. (n. d.). https://docs.nestjs.com/.

Getting started with yeoman. Yeoman. (n. d.). https://yeoman.io/learning/.

GlobalLogic Ukraine. Microservice arcitecture for the bigginers. (2023, March 16). https://www.globallogic.com/ua/insights/blogs/microservices-architecture-for-beginners-part-one/.

Hoang, C. (2023, December 27). *Monolith architecture*. Medium. https://tech.tamara.co/monolith-architecture-5f00270f384e.

Mijacobs, E. (n. d.). *What is infrastructure as code (IAC)? — azure DevOps*. Azure DevOps | Microsoft Learn. https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code.

Nagpal, A. (2024, July 30). *Monolithic vs microservices architecture: Advantages, disadvantages, and differences*. Medium. https://medium.com/@jasminepuno/monolithic-vs-microservices-architecture-advantages-disadvantages-and-differences-2bee6d1da8ca#:~:text=Monolithic%20architecture%20is%20a%20conventional,closely%20connected%20and%20centralized%20system.

Naik, A. (2020, June 17). *How to scaffold expressjs server and test it*. Medium. https://medium.com/craft-academy/how-to-scaffold-expressjs-server-and-test-it-d2a2ab1d30e0.

Node.js V22.8.0 documentation. Child process | Node.js v22.8.0 Documentation. (n.d.). https://nodejs.org/api/child_process.html.

Team, A. (2023, May 3). *What is scaffolding in Ruby on Rails?*. Altcademy Blog. https://www.altcademy.com/blog/what-is-scaffolding-in-ruby-on-rails/#:~:text=Scaffolding%20in%20Ruby%20on%20Rails%20is%20a%20helpful%20feature%20that,practices,%20and%20speeds%20up%20development.

What is a message queue? IBM. (2021, October 8). https://www.ibm.com/topics/message-queues.

What is an event broker? Solace. (2024, April 3). https://solace.com/what-is-an-event-broker/.

P. Kolinko

AUTOMATED GENERATION AND CONFIGURATION OF MICROSERVICES TO SIMPLIFY THE DEVELOPMENT PROCESS

The article "Automated Generation and Configuration of Microservices to Simplify the Development Process" explores how automated code generation and structured design streamline microservice-based application development. It introduces a custom software application built with Node.js, designed to automate the creation of a microservice architecture through scaffolding. This method simplifies the setup of core structures, speeding up development. The article starts with a comparison between monolithic and microservice architectures. In monolithic systems, all components—the user interface, business logic, and databases—are tightly integrated, making scaling and updates challenging as the system grows. Microservices break the application into independent services, allowing developers to scale, test, and deploy each part individually. This flexibility leads to improved fault tolerance and adaptability.

It also delves into microservice architecture layers, detailing how services communicate via lightweight protocols like HTTP and message queues. The importance of service discovery and load balancing, which ensure smooth communication in dynamic environments, is highlighted.

Node.js is chosen for its event-driven, non-blocking architecture, which is well-suited for handling multiple requests. Other technologies, such as Express.js, MongoDB, and Docker, further enhance scalability and efficiency. A key focus is on scaffolding, a technique that automates the generation of boilerplate code and project structure. This reduces repetitive tasks, ensures a consistent architecture, and improves development speed.

The article concludes by outlining the development of the custom application, showing how automation accelerates microservice creation, making the process more efficient and scalable.

Keywords: scaffolding, microservice architecture, code generation.

Матеріал надійшов 02.09.2024