MECHANISMS FOR IMPLEMENTING HEXAGONAL ARCHITECTURE IN NODE JS

Yura Abharian[1]

Received: 2021-09-03
Accepted: 2021-10-19
DOI: http://doi.org/10.46489/gpj.2021-1-3-11

Abstract. The mechanisms of realization of hexagonal architecture on Node js are substantiated in the article. It is emphasized that when developing an effective web application, it is important to configure the software architecture properly. A great way to build a supported web application is to create a flexible, extensible, and customizable architecture. The concept of hexagonal architecture as a popular architectural pattern in software development is defined. It is emphasized that this style of architecture facilitates the division of tasks, placing logic at different levels of the program. The hexagonal architecture aims to increase the maintainability of web applications so that the code requires less work in general. The scheme of the hexagonal architecture is presented, which is represented by a hexagon, where each of the sides of the hexagon represents different ways of interaction of the system with other systems. Examples of queries using HTTP-queries, REST API, SQL are offered. Each level of the hexagon depends on other layers, so the emphasis is on the ability to make individual changes without affecting the entire system. The structure of the hexagonal representation is described, where the program layer is represented as a hexagon. Inside the hexagon, there are domain objects and uses that work with them. It is emphasized that there are no initial dependencies. All dependencies point to the centre. The inside of a hexagon or domain does not depend on anything but itself. It ensures that business logic is separated from technical levels. It also ensures that it is possible to reuse domain logic. The kernel contains basic business logic and business rules. It is noted that outside the hexagon, there are various adapters that interact with the application. Different adapters will interact with different aspects of the program. The adapters on the left control the application because they call the core of the program. The application controls the adapters on the right because the program kernel calls them.

Keywords: hexagonal architecture, Node js, web application, multimodality, system, server.

[1] Yura Abharian, Specialist of the Department of Transport Technologies, SoftServe software developer, Zaporizhia National Technical University, Zhukovskoho Street, 64, 69063 Zaporizhia, Ukraine, ORCID: https://orcid.org/0000-0001-8519-2539

ВСТУП

Node (також відомий як Node.js, Nodejs, Node JS) About Node.js, and why you should add Node.js to your skill set?(2016) – це структура вводу-виводу, керована подіями, для двигуна JavaScript V8. Node.js призначений для написання масштабованих мережевих програм, таких як веб-сервери. Програми написані на JavaScript, використовуючи асинхронне введення-виведення, керовані подіями, щоб мінімізувати накладні витрати та максимізувати масштабованість. JavaScript заснований на реалізації від Google під час виконання – движку з влучною назвою «V8». Але, в той час як V8 підтримує переважно JavaScript у браузері (зокрема, Google Chrome), Node націлений на підтримку довготривалих серверних процесів.

На відміну від інших сучасних середовищ, процес Node не покладається на багатопоточність для підтримки одночасного виконання; він заснований на асинхронній моделі введення-виведення, керованої подіями. Його модель введення-виведення, керована подіями, що не блокує, а робить його легким та ефективним, ідеальним для додатків реального часу з інтенсивним використанням даних, які працюють на розподілених серверах.

Інтернет постійно розвивається і ставить перед розробниками веб-додатків ряд проблем. Великі інтернет-сервіси повинні мати справу з паралельністю в безпрецедентних масштабах. Високий трафік вимагає послуг для кращої обробки одночасних сеансів. Оскільки потреба в Інтернет-послугах зростає, для управління цим навантаженням необхідно використовувати нові методи проектування системи.

Існують дві поширені стратегії для обробки паралельності на веб-серверах: потоки та події. Веб-додатки реалізують потокову роботу, призначаючи кожен вхідний запит окремому потоку виконання. Наприклад, Apache використовує цей підхід у своєму мультимодульному робочому MPM. І навпаки, системи на основі подій використовують один потік для обробки подій, таких як вхідний запит, із черги. Кожен підхід має деякі особливі переваги та недолік (Titov, Doroshenko & Yatsenko, 2016).

Побудувати висококонкурентні системи за своєю суттю важко. Структурування коду для досягнення високої пропускної здатності погано підтримується існуючими моделями програмування. Хоча область потоків зазвичай використовується для вираження паралельності, високе використання ресурсів і обмеження масштабованості багатьох реалізацій потоків змусили розробників віддати перевагу підходу, орієнтованого на події (Titov, Doroshenko & Yatsenko, 2016; Prydatko et al., 2020), а архітектуру побудовану за принципом гексагональності (Konuklar, 2020). Node був створений Райаном Далем у 2009 році, а його зростання спонсорує Joyent. Початкова мета node полягала в тому, щоб реалізувати можливість створювати веб-сайти з можливостями push, як це видно в кількох веб-додатках (Gmail тощо). На Node впливають такі системи, як Ruby's Event Machine і Python's Twisted. Node розширює модель подій, представляючи цикл подій як мовну конструкцію, а не як бібліотеку.

Аналіз останніх досліджень і публікацій

Наукові набутки сучасних науковців у своїй більшості розкривають принципи формування гексагональної архітектури, як окремого механізму. Hrebeniuk & Pavlenko (2019) розкривають принципи формування архітектури портів та адаптерів в ітеративній розробці з обмеженнями по часу. Авторами розглянуто застосування архітектури портів та

адаптерів (інші назви цибулинна, шарувата, гексагональна) в ітеративній розробці програмного забезпечення відповідно до вимог, які надходять в хронологічному порядку на практичному прикладі. Кожна ітерація підкріплена схемою архітектури, виниклими проблемами і їх вирішенням. Показана доцільність застосування розглянутої архітектури при ітеративній розробці програмного забезпечення з обмеженнями по часу.

Prydatko, et al. (2020) висвітлили особливості архітектури та роботи інформаційно-довідкової системи швидкого доступу до бази даних навчального розкладу. Науковці подали модель клієнтської та серверної частин системи, описали особливості взаємодії клієнтської та серверної частин системи.

Naboishchykova & Osypenko (2020) описують загальні питання формування архітектури сучасних інформаційних систем. Із зарубіжних авторів варто відзначити такі роботи як: Ma (2020), Griffin (2020), Muhamad, Biyan & Abdulah, & Triayudi, Agung & Andrianingsih, Andrianingsih, Hexagonal-Driven Development, (n.d), Schlosser, Tobias & Friedrich, Michael & Kowerko, Danny, Abdulah & Triayudi (2020), Schlosser, Friedrich, & Kowerko (2019), Sunyaev (2020), Merzweiler, Angela & Stäubert, Sebastian & Strübing, Alexander & Tonmbiak, Armel & Kaulke, Knut & Drepper, Johannes & Bergh, Björn & Winter, Alfred (2021), Tsiperman, Grigoriy, Tsiperman (2021) Zhao, Jun (2021), Cao, Junwei & Wan, Yu-Xin & Tu, Guo-Yu & Zhang, Shu-Qing & Xia, Ai-Xuan & Liu, Xiao-Fei & Chen, Zhen & Lu, Chao (2014), Xuemin, Zhiming, & Ping (2012) та інші.

Проте, враховуючи описані наукові набутки, за темою, питання обґрунтування механізмів реалізації гексагональної архітектури на Node js залишається відкритим та потребує детального опрацювання.

Постановка завдання

Здійснити обґрунтування механізмів реалізації гексагональної архітектури на Node js.

РЕЗУЛЬТАТИ

Однією з найбільших помилок програмних додатків протягом багатьох років було проникнення бізнес-логіки в код інтерфейсу користувача. Даний факт викликає потрійну проблему: система не може бути акуратно протестована за допомогою автоматизованих наборів тестів, тому що частина логіки, яку необхідно протестувати, залежить від візуальних деталей, що часто змінюються, таких як розмір поля і розташування кнопок; неможливо перейти від використання системи, керованою людиною, до системи, що запускається партіями; важко або неможливо дозволити програмі керуватися іншою програмою, коли це стає необхідним.

Спроба рішення, повторювана у багатьох організаціях, полягає у створенні нового рівня в архітектурі з гарантією, що бізнес-логіка не буде поміщена на новий рівень.

У випадку якщо кожна частина функціональності, запропонована програмою, була доступна через API (програмний інтерфейс програми) або виклик функції, відділ тестування або контролю якості може запустити автоматизовані тестові сценарії для програми, щоб визначити, наявність нового кодування яке впливає на якість роботи. Бізнес-експерти можуть створювати автоматизовані тестові приклади до модернізації деталей графічного інтерфейсу, які повідомляють програмістам, про рівень якості виконаної роботи. Програма може бути розгорнута в «безголовому» режимі (тобто за відсутністю графічного інтерфейсу), тому доступний тільки API, та інші програми можуть використовувати його функціональні можливості – це спрощує загальний

дизайн складних наборів програм, а також дозволяє програмам-службам для бізнесу використовувати один одного без втручання людини через Інтернет. Нарешті, автоматичні функціональні регресійні тести виявляють будь-яке порушення щодо бізнес-логіки.

Аналогічна проблема існує на тому, що зазвичай вважається «іншою стороною» програми, де логіка програми прив'язується до зовнішньої бази даних або іншої служби. Коли сервер бази даних виходить з ладу або зазнає значної переробки або заміни, програмісти не можуть працювати, тому що їх робота пов'язана з наявністю бази даних. Це призводить до витрат на затримку та часто викликає невдоволення між людьми. Проте не очевидно, що ці дві проблеми пов'язані, але між ними існує симетрія, яка проявляється у характері розв'язання.

Архітектуру програм JavaScript є можливість розглядати з боку клієнтського додатку. Так клієнтську сторону веб-програми розглядають як саму програму, що визначається її верхніми межами, інтерфейсом користувача і з'єднанням зі стороною сервера (і аналогічними джерелами даних). Проте, на сьогодні інноваційним підходом до розробки веб-програми, який включає модульне тестування є гексагональна архітектура (яка не обов'язково має шість сторін).

При застосуванні принципів гексагональної архітектури, центральне ядро, що містить бізнес-логіку, оточене портами, які можуть бути поділені на дві категорії: порти з боку руху, де взаємодія ініціюється ззовні. Події DOM, які вказують на зворотні виклики, події, створювані платформою, та визначення setTimeout(), є портами на стороні руху; керовані бічні порти, де програма взаємодіє із зовнішньою інфраструктурою. Як приклад, розглянемо елементи DOM, якими можна маніпулювати, або де вставляється новий контент, або Ajax-дзвінки, призначені для клієнтської сторони, або навіть локальні бази даних.

Порти та адаптери, розглядаються, як головні складові інноваційного стилю архітектури, яка робить чіткий поділ між моделлю домену та пристроями, що використовуються для входів та виходів, схема такої архітектури наведена на рисунку 1.

Багаторівневі системи – це архітектурний стиль, що використовується, в основному, для запобігання сполучення, яке вважається найбільшим ворогом ремонтопридатності програмного забезпечення, особливо у великомасштабних системах. Чим більше елементів пов'язано, тим складніше змінити щось без ефекту пульсації, тестування стає складнішим, здатність розуміти і розмірковувати про вплив змін стає складнішим.

Порти та адаптери – це приклад багаторівневої архітектури, він відповідає всім обмеженням та властивостям багаторівневої системи.

Гексагональна архітектура виділяє у додатку три шари: домен (domain), додаток (application) та інфраструктура (framework): Домен. Шар містить основну бізнес-логіку. Він не повинен знати деталі реалізації зовнішніх шарів. Додаток. Шар діє як зв'язуюча ланка між шарами домену та інфраструктури. Інфраструктура. Реалізація взаємодії домену із зовнішнім світом. Внутрішні шари виглядають як чорний ящик.

Відповідно до цієї архітектури, з додатком взаємодіють два типи учасників: основні (driver) та вторинні (driven). Основні дійові особи надсилають запити та керують програмою (наприклад, користувачі або автоматизовані тести). Вторинні забезпечують інфраструктуру для зв'язку із зовнішнім світом (це адаптери бази даних, TCP- або HTTP-клієнти).

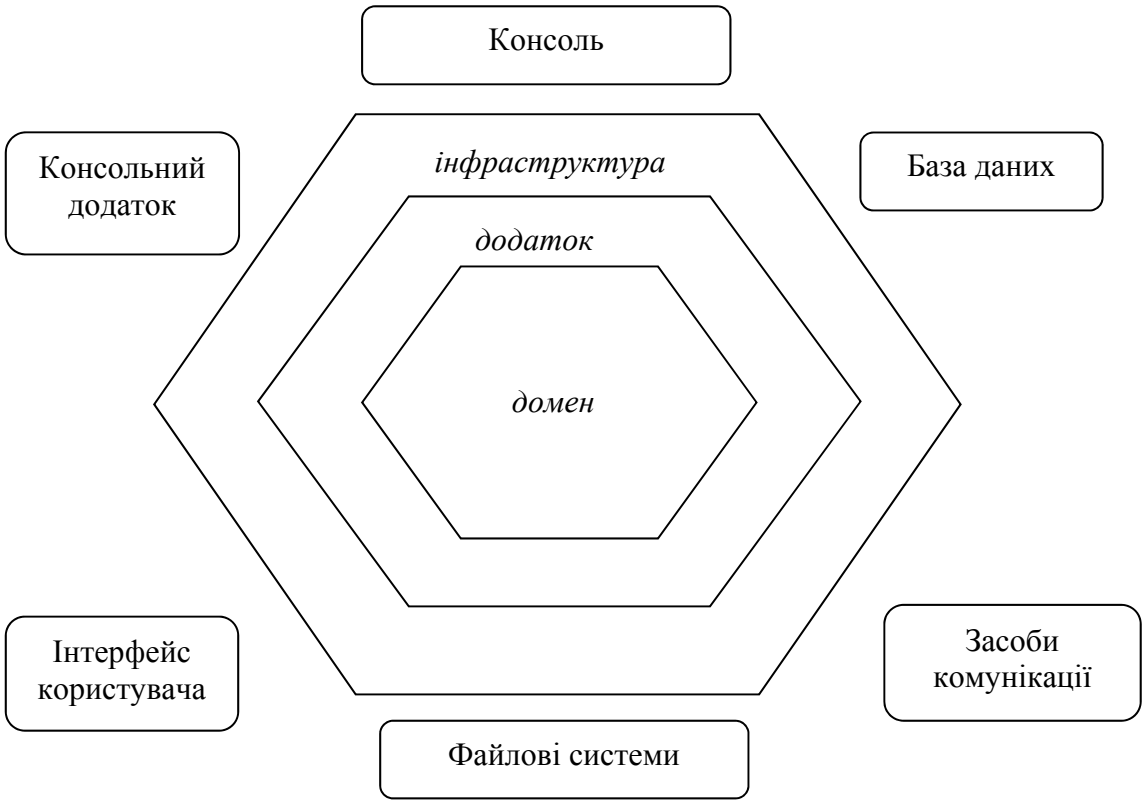

Рис. 1. Гексагональна архітектура додатка на Node js

Гексагональна архітектура має три рівні, ключовою частиною яких є модель предметної області, що містить всю логіку та правила застосування. Ніякі технологічні проблеми, наприклад контексти HTTP або виклики бази даних, не згадуються в домені, що дозволяє вносити зміни до технології, не торкаючись структури домену.

Навколо моделі предметної області перебуває рівень портів, який приймає всі запити, які відповідають варіанту використання, який керує роботою моделі предметної області. Шар портів є межею з доменом усередині та зовнішніми об'єктами зовні.

За межами портів знаходиться рівень адаптерів, технологічний стек, який приймає введення в якомусь форматі та робить висновок. Наприклад, за допомогою HTTP-запиту адаптер перетворює його на виклик на домен і маршалює відповідь з домену назад клієнту через HTTP. У адаптері немає доменної логіки; його єдина відповідальність – технічне перетворення між зовнішнім світом та областю. Будь-який адаптер, який дотримується протоколу порту, може використовувати його, і кілька адаптерів можуть використовувати той самий порт. Один з варіантів використання, це переключення на новий інтерфейс користувача зі старим і новим з використанням одних і тих же портів.

Тести гексагональної архітектури повинні бути зосереджені на поведінці, і, проводячи тестування безпосередньо з портом, при тестуванні не буде враховуватися будь-який інтерфейс користувача. Помилка багатьох розробників у тому, що вони тестують внутрішні деталі моделі предметної області. Це запобігає рефакторингу, оскільки зміна деталей реалізації призведе до збою тесту. Натомість модульне тестування слід проводити на

межі портів, який є загальнодоступним інтерфейсом, який залишається недоторканним навіть після зміни деталей реалізації.

Інтеграційні тести необхідні лише для тестування, наприклад конфігурації, наприклад, тестування зіставлення ORM для перевірки правильності конфігурації. Так само системні тести на зовнішньому кордоні складаються всього з декількох тестів впевненості, щоб переконатися, що все пов'язане разом, наприклад, що REST API працює, та аналогічні тести.

Ідея архітектури полягає в тому, щоб задовольнити кожен порт адаптером, що містить код, який є «проблематичним» для тестування; в цьому випадку труднощі виникають через посилання на фреймворки, Ajax і DOM. Ядро залишається вільним від зовнішніх залежностей і може бути протестовано детерміновано в одному процесі.

Вплив тестування на ведучу сторону відбувається за рахунок емалювання головних адаптерів. Головні порти дають відповідь на запит фреймворку або плагіну: зворотні відгуки, пов'язані з такими подіями, як клацання та зміна даних; природна еволюція, один об'єкт, що містить цей зворотний відгук, щоб вони могли взаємодіяти із загальним станом програмного додатку та системи загалом; у виродженому випадку один зворотний дзвінок створюється як анонімна функція.

Наприклад, jQuery ('# my_form'). Submit () дозволяє вказати зворотний зв'язок для виклику, коли користувач надсилає форму. Замість того, щоб вказувати його як анонімну функцію, його зазвичай легко параметризувати та тестувати незалежно від системних параметрів.

Керовані порти визначаються як об'єкти та функції, які необхідно замінити тестовими двійниками, щоб перевірити свою логіку ізольовано; наприклад, не робити реальний виклик Ajax, а підставляти запит функцією, яка повертає наперед визначене значення.

Загалом, є два варіанти: формування Test Double, який безпосередньо реалізує API, що викликається кодом (jQuery.get, jQuery.css або XMLHttpRequest.); обернення оригінальної бібліотеки та визначення власного інтерфейсу, який є змінюваним.

CONCLUSIONS

У роботі обґрунтовано механізми реалізації гексагональної архітектури на Node js. Гексагональна архітектура направлена на вирішення проблем, пов'язаних, наприклад, із традиційним нашаруванням, зв'язком та заплутуванням.

Таким чином, чим більше логіки у JavaScript, тим більше повинно бути циклів тестування незалежно від усього, що відбувається на стороні клієнта. Еволюція у напрямку моделювання портів, Ajax та DOM є природною; перспективним розвитком є використання адаптерів для додаткової ізоляції ядра від зовнішніх проблем, надаючи йому API замість потоку викликів платформи.

References

About Node.js, and why you should add Node.js to your skill set? (2016). http://blog.training.com/2016/09/about-nodejs-and-why-you-should-add.html

Titov, D. S., Doroshenko, A. Y., & Yatsenko, O. A. (2016). Automated development of a parallel distributed system for streaming data processing. *PROBLEMS IN PROGRAMMING*, (2-3), 096–104. https://doi.org/10.15407/pp2016.02-03.096

Konuklar, T. (2020, October 12). *Hexagonal (ports & adapters) architecture - idealo tech blog - medium*.

Idealo Tech Blog. https://medium.com/idealo-tech-blog/hexagonal-ports-adapters-architecture-e3617bcf00a0

Hrebeniuk, O., & Pavlenko, V. (2019). Arkhitektura portiv ta adapteriv v iteratyvnii rozrobtsi z obmezhenniamy po chasu [Architecture of ports and adapters in iterative development with time constraints]. *Visnyk Universytetu «Ukraina» Seriia Informatyka, obchysliuvalna tekhnika ta kibernetyka*, *1*(22). https://visn-it.uu.edu.ua/index.php/visn-icct/article/view/24

Prydatko, O. V., Burak, N. Ye., Dzen, V. Ye., & Kunynets, M. S. (2020). Adaptive information-references system UniBell as part of the Smart-university project. *Scientific Bulletin of UNFU*, *30*(5), 113–121. https://doi.org/10.36930/40300518

Naboishchykova, Ye. O., & Osypenko, V. V. (2020). Arkhitektura informatsiinykh system [Information systems architecture]. In *Elektromekhanichni ta informatsiini systemy*: materialy Vseukrainskoi naukovo-praktychnoi Internet-konferentsii molodykh uchenykh ta studentiv, prysviachena 90-y richnytsi zasnuvannia Kyivskoho natsionalnoho universytetu tekhnolohii ta dyzainu, m. Kyiv, 21 kvitnia 2020 roku. Kyiv : Kyivskyi natsionalnyi universytet tekhnolohii ta dyzainu, S. 100-101.

Ma, L. (2020). Multimedia Simulation-Based Architecture CAD System Model. *Computer-Aided Design and Applications*, *18*(S1), 53–64. https://doi.org/10.14733/cadaps.2021.s1.53-64

Griffin, J. (2020). Hexagonal-Driven Development. In *Domain-Driven Laravel* (pp. 521–544). Apress. https://doi.org/10.1007/978-1-4842-6023-4_17

Abdulah, M. B., & Triayudi, A. (2020). Efek behavior-driven development, hexagonal architecture dan event sourcing terhadap pengembangan perangkat lunak bisnis. *Jurnal Mantik Penusa*, *24*(1), 1-5.

Schlosser, T., Friedrich, M., & Kowerko, D. (2019). Hexagonal Image Processing in the Context of Machine Learning: Conception of a Biologically Inspired Hexagonal Deep Learning Framework. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. IEEE. https://doi.org/10.1109/icmla.2019.00300

Sunyaev, A. (2020). Information Systems Architecture. In *Internet Computing* (pp. 25–49). Springer International Publishing. https://doi.org/10.1007/978-3-030-34957-8_2

Sunyaev, A. (2020). Design of Good Information Systems Architectures. In *Internet Computing* (pp. 51–81). Springer International Publishing. https://doi.org/10.1007/978-3-030-34957-8_3

Merzweiler, A., Stäubert, S., Strübing, A., Tonmbiak, A., Kaulke, K., Drepper, J., Bergh, B., & Winter, A. (2021). The Process of Modeling Information System Architectures with IHE. In *German Medical Data Sciences: Bringing Data to Life*. IOS Press. https://doi.org/10.3233/shti210065

Tsiperman, G. (2021). Design techniques for a seamless information system architecture. *arXiv preprint arXiv:2102.10830*.

Zhao, J. (2021). Cloud Based Information System Architecture in Construction Site. In *2021 International Conference on Applications and Techniques in Cyber Intelligence* (pp. 483–487). Springer International Publishing. https://doi.org/10.1007/978-3-030-79200-8_73

Cao, Junwei & Wan, Yu-Xin & Tu, Guo-Yu & Zhang, Shu-Qing & Xia, Ai-Xuan & Liu, Xiao-Fei & Chen, Zhen & Lu, Chao. (2014). Information System Architecture for Smart Grids. *Chinese Journal of Computers*, *36*(1), 143–

167. https://doi.org/10.3724/sp.j.1016.2013.00143

Xuemin, Z., Zhiming, S., & Ping, G. (2012). The Process of Information Systems Architecture Development. *Procedia Engineering*, *29*, 775–779. https://doi.org/10.1016/j.proeng.2012.01.040