

МЕТОДИ ТА ЗАСОБИ СИСТЕМИ ПЕРЕВІРКИ СУМІСНОСТІ ОКРЕМИХ КОМПОНЕНТ ВЕБСЕРВЕРІВ

Дмитро Крупа

Національний університет “Львівська політехніка”,
кафедра інформаційних систем та мереж, Львів, Україна
E-mail: dmytro.v.krupa@lpnu.ua, ORCID: 0009-0002-6087-9988

© Крупа Д., 2024

У статті розглянуто сучасний стан проблем у роботі з API різних систем. Проаналізовано найпоширеніші методології (Agile та DevOps), методи та засоби побудови автоматизованих конвеєрів для збірки та тестування продуктів проєкту; наведено загальний вигляд конвеєра, що слугує відправною точкою під час розроблення проєктів. У результаті аналізу опитувань розробників DevOps інженерів виділено ключові проблеми в інтеграції між різними системами з використанням API.

У роботі описано повний процес виконання тестів, що передбачає ініціалізацію кроку тестування, варіанти побудови конвеєра, опрацювання виключних ситуацій та збереження результатів. Наведено взаємодію різних зацікавлених сторін. Зазначені процеси проілюстровані діаграмами послідовностей, прецедентів та діяльності. Наведено псевдокод алгоритму, що описує використання нечіткого логічного виведення для отримання рекомендацій на основі тестування. Розроблено розширений контракт для тестування, який передбачає: пари запит-відповідь, обмеження сфери значень полів та предикати для забезпечення логічної сумісності компонентів. Використання цього контракту дає змогу не лише перевіряти сумісність API компонент вебсерверів, але й задавати правила логічної відповідності між об'єктами. Впровадження онтологічного підходу для тестування створює додатковий рівень тестів, що перевіряють відповідність між сутностями системи-споживача та онтологією системи-постачальника. Впровадження перевірки сфери значень полів та їхніх типів під час виконання тестів дасть змогу надати розробникові на етапі тестування повноту інформації про можливі проблеми під час роботи двох систем. Оскільки онтологія повинна постійно підтримуватися експертами в галузі, то завдяки використанню розширених контрактних тестів можна буде зменшити ймовірність помилок унаслідок недостатньої комунікації через зміну концептів у бізнес-структурі системи-постачальника. Розроблений метод передбачає використання нечіткої логіки для прийняття рішень на основі нечітких оцінок сумісності двох компонент.

Ключові слова: онтологічний підхід, автоматизоване тестування, сумісність компонента, контрактне тестування, DevOps.

Вступ

Згідно зі статистикою використання Agile та DevOps методологій розподіляється як 47 % та 37 % відповідно. Зазначимо також, що ці методології можуть об'єднуватися в межах одного проєкту для підвищення рівня ефективності роботи команди. Основною складовою DevOps методології з технічної точки зору є CI/CD, що безпосередньо впливає на сталість роботи продукту [1]. Зосередження на CI/CD допомагає забезпечити те, що програмний продукт є якісним та стабільним упродовж усього циклу розроблення та може бути переданий (розгорнутий) для кінцевих користувачів у будь-який момент. Однією зі складових зазначених процесів є автоматизовані

тести, що зменшують кількість роботи, яку повинна виконувати команда тестувальників перед кожним релізом. Із розвитком інформаційних технологій збільшується кількість сторонніх сервісів, з якими необхідно інтегруватися. Особливо це стосується CRM та ERP систем, що відповідають за операційні процеси компанії і прямо впливають на її прибутки. Зміна API системи-постачальника може спричинити затримку в релізах та необхідність термінового розгортання виправлень (hotfix), що може збільшити терміни виконання проекту.

Згідно з дослідженнями компанії Postman Inc 2023 року (Postman 2023 State of the API Report), найбільшою складністю для 52 % респондентів була недостатня кількість документації. Також у топ-3 проблем входять складнощі дослідження API систем (32 %) та нестача часу (27 %) [2]. Це вказує на те, що коли API системи-постачальника змінюється, то зміни складно відстежити, особливо, коли вони стосуються внутрішньої функціональності.

Формулювання проблеми

Основною проблемою є реагування на зміни у бізнес-логіці системи-постачальника. Зазвичай великим релізам передують кількарізкові нагадування (наприклад, як це робить Google для своїх сервісів), але зміни від компаній, що надають API своїх CRM та ERP, можуть бути пропущені через недостатню комунікацію. Стандартний контракт для контрактного тестування містить лише дані для перевірки структури запиту та відповіді: шлях до ресурсу, метод запиту, об'єкт запиту, об'єкт відповіді, статус код відповіді – без урахування логічного навантаження [13]. Окрім того, не перевіряється обмеження на сферу визначення полів запиту та відповіді, що може призвести до помилок у роботі систем, коли дані вводять користувач.

Аналіз останніх досліджень та публікацій

Процес тестування інтеграцій між системами зазвичай є частиною усього плану тестування команди перед розгортанням продукту для користувачів. Тому розглянемо місце таких тестів у Agile та DevOps методологіях та засоби, за допомогою яких вони реалізуються.

На сьогодні де-факто стандартом розроблення програмних продуктів є Agile або DevOps методології. Через часте їхнє змішування виникають труднощі в їхньому розділенні і чіткому визначенні, якої саме методології дотримуються під час реалізації проекту [3]. DevOps має два важливі аспекти – методологія та підхід (процес) [4]. DevOps як методологія відповідає за взаємодію між командами, і його метою є максимальна автоматизація процесів та покращення співпраці. DevOps як підхід – це конкретні кроки, які реалізують DevOps методологію та передбачають: безперервну інтеграцію, безперервну доставку та моніторинг. Поетапний DevOps процес зображено на рис. 1. Нечіткість у визначенні меж між цими методологіями часто стає причиною того, що проєкт визначають як “Agile з DevOps практиками”. Насправді для перевірки компонент на сумісність немає значення, яке їхнє місце в методології проєкту. Важливими є наявність тестів та регулярність їхнього виконання. Отже, незважаючи на обрану методологію проєкту, єдиним місцем для реалізації автоматизованих тестів перевірки інтеграції між системами є CI/CD процес.

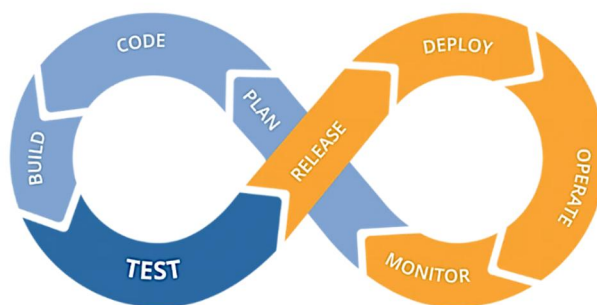


Рис. 1. DevOps процес

Фактично кожний проєкт, що є достатньо складним, має побудований той чи той CI/CD процес. CI/CD – це набір практик, що забезпечують автоматизовану збірку, перевірку, тестування та розгортання продукту [5]. Одним із найвідоміших засобів створення CI/CD конвеєрів є Jenkins (близько половини респондентів користуються ним) [6]. Розгляньмо найбільш гнучкий метод створення конвеєрів – Jenkins Pipeline. Jenkins Pipeline (здебільшого Pipeline) – це набір плагінів, що дає змогу реалізувати CI/CD процеси проєкту у середовищі Jenkins [7–8]. Його недоліком є високий поріг входження, адже для написання скриптів необхідно добре розуміти логіку роботи конвеєрів. Приклад Pipeline з документації Jenkins наведено на рис. 2.

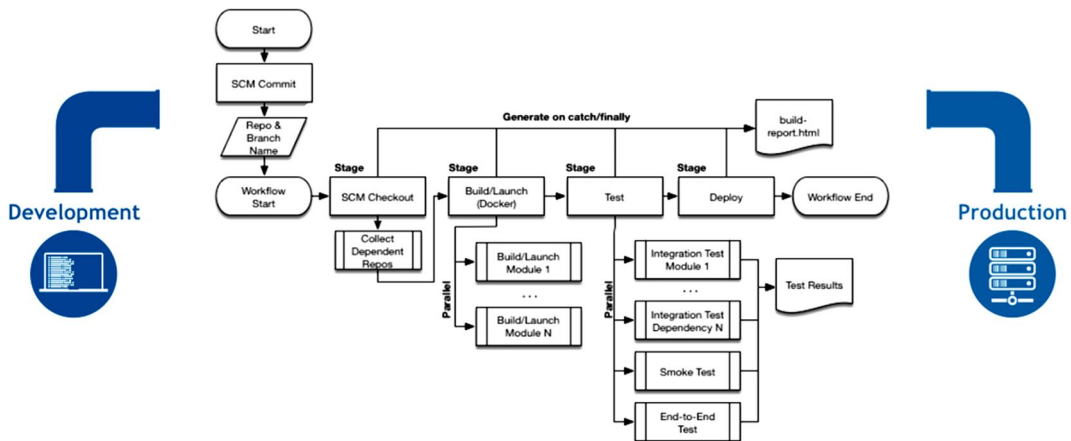


Рис. 2. Загальний вигляд Jenkins Pipeline

Зауважимо, що, вживаючи “CI/CD”, часто мають на увазі “Jenkins Pipeline” або “Jenkins script”, але CI/CD – це підхід та опис безперервної інтеграції та розгортання, а Jenkins Pipeline – це реалізація даного підходу. Для запуску CI/CD здебільшого використовуються тригери, адже ручний виклик процесу суперечить ідеї автоматизації. Серед тригерів можна виділити: запуск до/після іншого конвеєру, cron, HTTP/S запит та webhook. Переважно стандартний конвеєр розпочинає роботу при push-події у SCM, збирає, тестує та розгортає код, що був завантажений розробником у SCM систему. Результати тестів можуть відображатися лише в консолі або як звіт. Окрім того, вони можуть використовуватися як контрольна точка якості. Кожний CI/CD процес будується під конкретний проєкт, враховуючи його вимоги, особливості, бюджет, план випуску версій, архітектуру та інші характеристики.

Досі не існує інструменту, призначеного для валідації онтологій двох різних систем або для перевірки відповідності онтології структурі однієї з систем. Існують такі інструменти, як-от Protégé, що можуть валідувати онтологію окремо від бізнес-системи, використовуючи reasoner-и, та SHACL – для валідації структурних вимог онтології [14–15]. Для перевірки якості онтологій використовують OntoQA, що має змогу оцінити зв’язність, сумісність та повноту онтології. Цей інструмент уперше був описаний 2005 року, а станом на зараз проєкт давно не отримував оновлень та може бути несумісний із сучасними засобами розроблення [16–17].

Формулювання цілі статті

Ціллю статті є визначити місце тестування сумісності компонент у структурі DevOps процесів та запропонувати розширення стандартного контрактного тестування для можливості перевірки логічної сумісності між двома компонентами вебсерверів та врахування обмежень допустимих значень полів.

Підціллю статті є описати такий метод тестування, який не буде накладати обмежень під час розроблення, а буде відповідати за якість роботи продукту, що уже надається користувачам. Необхідно звернути увагу на максимальну автоматизацію виконання та викликів даних тестів та їхню інтеграцію із сучасними методами та засобами розроблення тестів.

Виклад основного матеріалу

Оскільки більшість проєктів використовують Agile або DevOps методології із створеними CI/CD процесами, очевидно, що єдиним місцем, де можуть існувати тести перевірки сумісності компонент, є крок “Test” у CI/CD. Але цей підхід передбачає виклик тестів лише тоді, коли розробники додають зміни у код у системі контролю версій (SCM system). Для досягнення можливості перевірки інтеграції як з боку системи-постачальника, так і з боку системи-споживача, необхідно ввести можливість виклику тестів сторонніми командами. Безсумнівним рішенням є надати кінцевий пристрій, що захищається, для виклику тестів. Такий підхід є максимально простим у реалізації, адже забезпечує право виклику тестів зовнішній команді, тоді як внутрішня команда займається розробленням та підтримкою тестів. Окрім того, команді розробників системи-постачальника, щоб тести працювали стабільно, необхідно розглянути такі сценарії:

1. Паралельне виконання тестів.
2. Відсутність файлу, що містить онтологію.
3. Додаткові канали комунікації.
4. Запуск тестів не лише на вимогу, а як заплановане завдання (cron-завдання).

На рис. 3 зображено діаграму послідовності вищенаведеного процесу тестування. Її загальний вигляд передбачає 4 кроки:

1. Розпочати перевірку – дія (тригер), що розпочинає процес тестування. Ними можуть бути: запит на кінцевий пристрій, що захищається, ручний запуск процесу у системі, cron-триггер.

2. Отримати запит на онтологію та структуру – запит на отримання файлів із онтологією та запитами від системи-постачальника.

3. Онтологія та структура – власне файли із онтологією та файли із структурою запитів. Для прикладу, це можуть бути .owl та .json формати.

4. Повідомити – надіслати повідомлення у канал комунікації (Slack, Email, SMS). Рекомендовано використовувати кілька каналів для високої надійності, а також розділяти їх для різних команд або залежно від рівня важливості повідомлення.

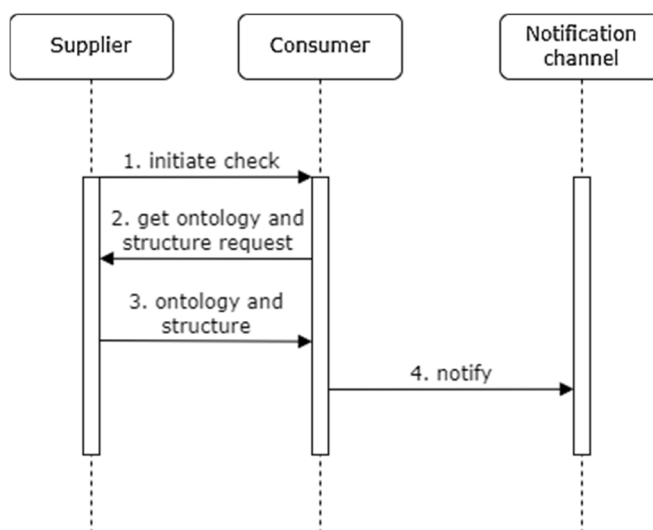


Рис. 3. Діаграма послідовності

Використаймо діаграму прецедентів для формалізації типів ролей, що зацікавлені у тестуванні сумісності компонент та те, як вони взаємодіють із системою (див. рис. 4). Актор “Supplier system” відображає систему-постачальника, або її розробників, які ініціюють перевірку викликом кінцевого пристрою, що захищається. “Engineer” – це розробник системи-споживача, який має мати можливість не лише запуснути тести, а й переглянути статистику. Використання статистики не є

необхідним кроком для реалізації цього виду тестування, але більшість команд стараються збирати статистику для формування звітів про роботу під час виконання проекту чи однієї ітерації (спринта). У разі використання статистики, вона повинна зберігатися у базі даних (актор “Database”) та бути доступною виключно для команди розробників.

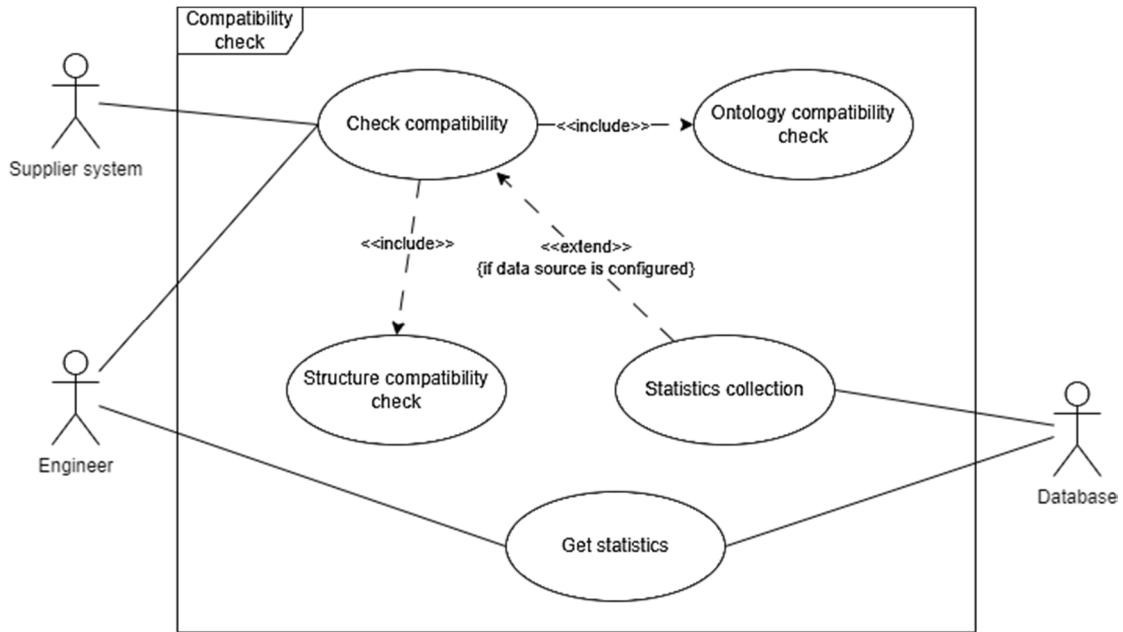


Рис. 4. Діаграма прецедентів

Власне процес перевірки сумісності компонент зображено на діаграмі діяльності (див. рис. 5). Більшість кроків є цілком зрозумілими: отримати правила з власної системи, із системи постачальника, оцінити сумісність між системами. Єдиним місцем, яке можна змінити відповідно до вимог, – це результат виконання перевірки, чи всі правила існують.

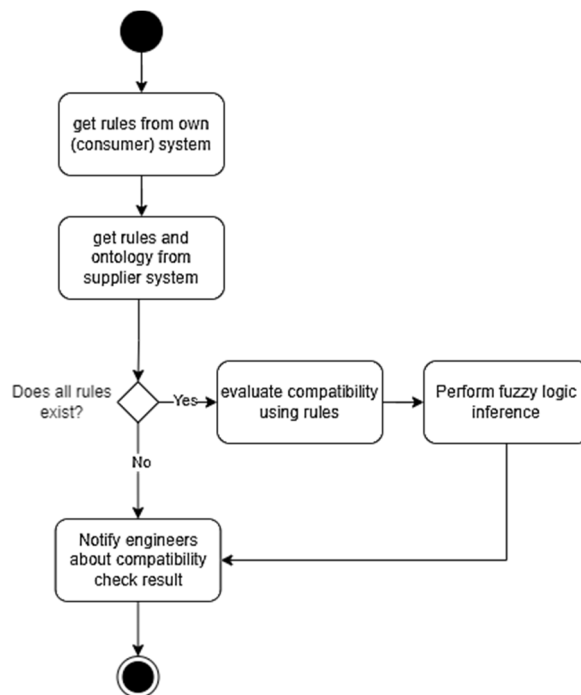


Рис. 5. Діаграма діяльності

Відкритим залишається питання, що робити у разі, коли неможливо завантажити файл з онтологією або неможливого його відкрити. Є кілька варіантів вирішення цієї проблеми, але рішення повинно прийматися разом із менеджментом проєкту та експертами у галузі для врахування всіх ризиків:

1. Використання попередньої онтології (можливо за умови, що онтологія була завантажена хоча б раз).
2. Термінова комунікація зі службою підтримки системи-постачальника.
3. Пропуск тестів перевірки сумісності без урахування їх у результаті виконання всього пакету тестів.

Для правильного та повного виконання тестів було розширено стандартний контракт та додано до нього перевірку на сферу визначення полів та перевірку логічної сумісності між бізнес-структурою системи-споживача та онтологією системи-постачальника. Цей крок є необхідним для досягнення кращої якості тестування сумісності, беручи до уваги контекст пари запит-відповідь, а не лише загальну структуру. Також розроблюваний метод тестування надаватиме інформацію про зміни у структурі онтології системи-постачальника, яка зазвичай підтримується за допомогою експертів у галузі, що дає змогу завжди використовувати найактуальніші дані.

Розширена версія контракту може бути представлена у файлі будь якого формату, що є зручним для опрацювання модулями тестування. Структура файлу та опис наведено у таблиці 1.

Таблиця 1

Структура файлу контракту

Назва поля	Опис	Приклад
url	Адреса запиту	/api/user/create
Method	HTTP метод	POST
Response status code	Код відповіді у разі успішного запиту	200 (OK)
Requrst-Response pair	Пара запит – відповідь, використовується для перевірки правильності опрацювання даних	Запит {"id" : null, "name" : "Dmytro"} Відповідь: {"id" : 3132, "name" : "Dmytro"}
Field types conditions	Масив типів даних усіх або деяких полів запиту/відповіді	{"age" : "int"}
Logical conditions	Умова для забезпечення сумісності онтології та сутностей (бізнес-об'єктів)	"User is Person"
Limitation of permissible values	Масив допустимих значень усіх або деяких полів запиту/відповіді. Може містити правила виду {x > 100}, { ∃Y }	{"age > 100"}

Кожний тест повинен перевірити, чи збігаються об'єкти запитів і чи не порушується наперед визначена умова відповідності бізнес-об'єкту системи-постачальника (представляють концептом онтології). Система також використовуватиме нечітке логічне виведення для надання рекомендацій чи є критично необхідно вносити виправлення чи ні [9–10]. В алгоритмі беруть участь дві пари об'єктів – клас-сутність (entity) і DTO із боку системи-споживача та концепт і DTO з боку системи-постачальника. Порівняння DTO класів між системами відповідає класичному контрактному тестуванню, а порівняння класу-сутності та концепту є схожим процесом, але з використанням додаткових класів та правил для опису типу зв'язку між ними [11–12]. Контракти та всі додаткові правила, що описують взаємозв'язок між класами та концептами, об'єднаємо в одне поняття як "розширений контракт". Псевдокод алгоритму подано нижче.

Псевдокод:

Ініціалізуємо масив розширених контрактів A для перевірки сумісності компонент.

Для кожної пари перевірок:

перевіряємо виконання очікуваних кодів відповідей між об'єктами-запитами;

перевіряємо кількість обов'язкових полів об'єктів-запитів;

якщо коди відповідей не збігаються **або** кількість обов'язкових полів не збігається записати дані перевірки у звіт.

Кінець, якщо:

перевіряємо кількість та типи полів між entity та концептом;

ініціалізуємо змінну *оцінка основних властивостей* відповідно до функцій приналежності;

ініціалізуємо змінну *оцінка додаткових властивостей* відповідно до функцій приналежності;

ініціалізуємо змінну *оцінка відповідності онтології* відповідно до функцій приналежності;

застосовуємо правила.

Агрегуємо результати для кожного виходу.

Конвертуємо нечіткі результати у чіткі значення.

Кінець для кожної.

Сформувати звіт та надіслати його у канал комунікації.

Якщо є під'єднання до бази даних то:

зберегти звіт у базі даних.

Кінець, якщо

Результат виконання тестів може бути використаний для перевірки контрольних точок якості і згідно зі стандартним DevOps процесом (або процесом ітерацій у Agile/Scrum) наступним кроком буде або випуск (реліз) нової версії продукту, або виправлення помилок. Якщо ж тести були запущені зі сторони системи-постачальника і були виявлені помилки, то необхідно перепланувати поточну ітерацію з урахуванням помилок, які треба виправити, та плану релізу системи-постачальника.

Висновки

Результатом дослідження є аналіз місця виконання автоматизованих контрактних тестів для перевірки сумісності окремих компонент вебсерверів у структурі Agile/DevOps процесів та розроблення розширених контрактів для перевірки структури, типів, обмежень та логічного зв'язку між компонентами вебсерверів. Оскільки над різними складовими комплексної системи часто працюють різні команди, то неузгоджені релізи можуть порушити стабільність роботи як цілої системи, так і окремої компоненти. Для уникнення проблем із комунікацією між командами та отримання результатів сумісності компонент був розроблений проактивний метод реагування на зміни у системі-постачальнику шляхом можливості виклику тестів сторонньою командою. Такий підхід був продемонстрований діаграмами, що описують процес виклику тестів та процес їхнього виконання. Крім того, був наведений псевдокод алгоритму перевірки систем на сумісність із використанням нечіткого логічного виведення для надання рекомендацій розробникам та керівникам проєктів. Перспективою подальших досліджень є розроблення програмних модулів та інтерфейсів для реалізації системи перевірки сумісності окремих компонент вебсерверів.

СПИСОК ЛІТЕРАТУРИ

1. (2024, May 29). *DevOps Market Statistics – Key Figures and Trends in 2024*. TechReport. <https://techreport.com/statistics/software-web/devops-market-statistics-2024/>
2. (2023). *State of the API Report*. Postman. <https://www.postman.com/state-of-api/api-global-growth/#api-global-growth>
3. (n.d.). *How Do Agile and DevOps Interrelate?* Instatus. <https://instatus.com/blog/agile-devops>

4. Sharma, S., Sarkar, D., Gupta, D. (2012, December). Agile Processes and Methodologies: A Conceptual Study. *International Journal on Computer Science and Engineering*, 4(5).
5. MAAYAN, G. D. (2023, December 15). *The Future of Jenkins in 2024*. DevOps.com. <https://devops.com/the-future-of-jenkins-in-2024/>
6. (n.d.). *Pipeline*. Jenkins. <https://www.jenkins.io/doc/book/pipeline/>
7. (2024, April 19). *How to Make a CI-CD Pipeline in Jenkins?* GeeksforGeeks. <https://www.geeksforgeeks.org/how-to-make-a-ci-cd-pipeline-in-jenkins/>
8. Sotomayor, J., Allala, S., Santiago, D., King, T. (2022). Comparison of open-source runtime testing tools for microservices. *Software Quality Journal*, 31(1), 1–33. <https://doi.org/10.1007/s11219-022-09583-4>
9. Желдак Т. А., Коряшкіна Л. С. Коряшкіна. (2020). *Нечіткі множини в системах управління та прийняття рішень*. НТУ “Дніпровська політехніка”.
10. Cingolani, P., Alcal, J. (2013). FuzzyLogic: A Java library to design fuzzy logic controllers according to the standard for fuzzy control programming. *International Journal of Computational Intelligence Systems*.
11. Gillis, A. S. (2023, March). *API testing*. TechTarget. <https://www.techtarget.com/searchapparchitecture/definition/API-testing>
12. Braakman, W. (2023, March 30). *Introduction to Contract Testing*. Medium. <https://www.techtarget.com/searchapparchitecture/definition/API-testing>
13. (n.d.). *Writing consumer tests*. GitLab. https://docs.gitlab.com/ee/development/testing_guide/contract/consumer_tests.html
14. (n.d.). *Reasoner PReferences*. Protégé 5 Documentation. <https://protegeproject.github.io/protege/pReferences/reasoner/>
15. World Wide Web Consortium (2017, July 20). *Shapes Constraint Language (SHACL)*. W3C. <https://www.w3.org/TR/shacl/>
16. Tartir, S., Arpinar, I. B., Moore, M., Sheth, A. P., Aleman-Meza, B. (2005). OntoQA: Metric-Based Ontology Quality Analysis. *IEEE ICDM 2005 Workshop on Knowledge Acquisition from Distributed, Autonomous, Semantically Heterogeneous Data and Knowledge Sources*. https://www.researchgate.net/publication/266795541_OntoQA_Metric-Based_Ontology_Quality_Analysis
17. Tartir, S. (2020, May 7). *OntoQA*. GitHub. <https://github.com/Samir-Tartir/OntoQA>

REFERENCES

1. (2024, May 29). *DevOps Market Statistics – Key Figures and Trends in 2024*. TechReport. <https://techreport.com/statistics/software-web/devops-market-statistics-2024/>
2. (2023). *State of the API Report*. Postman. <https://www.postman.com/state-of-api/api-global-growth/#api-global-growth>
3. (n.d.). *How Do Agile and DevOps Interrelate?* Instatus. <https://instatus.com/blog/agile-devops>
4. Sharma, S., Sarkar, D., Gupta, D. (2012). Agile Processes and Methodologies: A Conceptual Study. *International Journal on Computer Science and Engineering*, 4(5).
5. MAAYAN, G. D. (2023, December 15). *The Future of Jenkins in 2024*. DevOps.com. <https://devops.com/the-future-of-jenkins-in-2024/>
6. (n.d.). *Pipeline*. Jenkins. <https://www.jenkins.io/doc/book/pipeline/>
7. (2024, April 19). *How to Make a CI-CD Pipeline in Jenkins?* GeeksforGeeks. <https://www.geeksforgeeks.org/how-to-make-a-ci-cd-pipeline-in-jenkins/>
8. Sotomayor, J., Allala, S., Santiago, D., King, T. (2022). Comparison of open-source runtime testing tools for microservices. *Software Quality Journal*, 31(1), 1–33. <https://doi.org/10.1007/s11219-022-09583-4>
9. Zheldak, T. A., Koryashkina, L. S. (2020). *Fuzzy sets in control and decision-making systems*. NTU “Dnipro Polytechnic”.
10. Cingolani, P., Alcal, J. (2013). FuzzyLogic: A Java library to design fuzzy logic controllers according to the standard for fuzzy control programming. *International Journal of Computational Intelligence Systems*.
11. Gillis, A. S. (2023, March). *API testing*. TechTarget. <https://www.techtarget.com/searchapparchitecture/definition/API-testing>
12. Braakman, W. (2023, March 30). *Introduction to Contract Testing*. Medium. <https://www.techtarget.com/searchapparchitecture/definition/API-testing>

13. (n.d.). *Writing consumer tests*. GitLab. https://docs.gitlab.com/ee/development/testing_guide/contract/consumer_tests.html
14. (n.d.). *Reasoner PReferences*. Protégé 5 Documentation. <https://protegeproject.github.io/protege/pReferences/reasoner/>
15. World Wide Web Consortium (2017, July 20). *Shapes Constraint Language (SHACL)*. W3C. <https://www.w3.org/TR/shacl/>
16. Tartir, S., Arpinar, I. B., Moore, M., Sheth, A. P., Aleman-Meza, B. (2005). *OntoQA: Metric-Based Ontology Quality Analysis*. *IEEE ICDM 2005 Workshop on Knowledge Acquisition from Distributed, Autonomous, Semantically Heterogeneous Data and Knowledge Sources*. https://www.researchgate.net/publication/266795541_OntoQA_Metric-Based_Ontology_Quality_Analysis
17. Tartir, S. (2020, May 7). *OntoQA*. GitHub. <https://github.com/Samir-Tartir/OntoQA>

METHODS AND TOOLS FOR THE SYSTEM OF COMPATIBILITY CHECKING OF INDIVIDUAL COMPONENTS OF WEB-SERVERS

Dmytro Krupa

Lviv Polytechnic National University,
Department of Information Systems and Networks, Lviv, Ukraine
E-mail: dmytro.v.krupa@lpnu.ua, ORCID: 0009-0002-6087-9988

© Krupa D., 2024

This article examines the current state of issues in working with APIs of various systems. The most common methodologies (Agile and DevOps), methods, and tools for building automated pipelines for project assembly and testing were analyzed, and a general pipeline structure was presented, which serves as a starting point for project development. As a result of analyzing surveys of developers and DevOps engineers, key problems in integration between different systems using APIs were identified.

The paper describes the complete process of executing tests, which includes the initialization of the testing step, pipeline construction options, exception handling, and result storage. The interaction of various stakeholders is presented. These processes are illustrated with sequence, use case, and activity diagrams. Pseudocode of an algorithm is provided, which describes the use of fuzzy logic inference for generating recommendations based on testing. An extended contract for contract testing was developed, which includes request-response pairs, field value range constraints, and predicates to ensure the logical compatibility of components. The use of this contract allows not only to verify the compatibility of API components of web servers but also to define rules for logical consistency between objects. The implementation of an ontological approach for testing creates an additional layer of tests that check the correspondence between the entities of the consumer system and the ontology of the provider system. The introduction of field value range and type checks during test execution will provide the developer with comprehensive information about potential issues when two systems interact. Since the ontology must be continuously maintained by domain experts, the use of extended contract tests will reduce the likelihood of errors due to insufficient communication caused by changes in the concepts within the provider system's business structure. The developed method involves the use of fuzzy logic for decision-making based on fuzzy assessments of the compatibility of two components

Keywords: ontological approach, automated testing, component compatibility, contract testing, DevOps.