Issue 16, 2024

УДК 004.05

МЕТОДИ ТА ЗАСОБИ СИСТЕМИ ПЕРЕВІРКИ СУМІСНОСТІ ОКРЕМИХ КОМПОНЕНТ ВЕБ-СЕРВЕРІВ

Крупа Дмитро

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

У цій статті розглянуто сучасний стан проблем у роботі з API різних систем. Були проаналізовані найпоширеніші методології (Agile та DevOps), методи та засоби побудови автоматизованих конвеєрів для збірки та тестування продуктів проекту, наведено загальний вигляд конвеєра, що слугує відправною точкою при розробці проектів. У результаті аналізу опитувань розробників, DevOps інженерів були виділені ключові проблеми в інтеграції між різними системами з використанням API.

У роботі описано повний процес виконання тестів, що включає в себе ініціалізацію кроку тестування, варіанти побудови конвеєра, опрацювання виключних ситуацій та збереження результатів. Наведена взаємодія різних зацікавлених сторін. Дані процеси проілюстровані діаграмами послідовностей, прецедентів та діяльності. Наведено псевдокод алгоритму, що описує використання нечіткого логічного виведення для отримання рекомендацій на основі тестування. Було розроблено розширений контракт для тестування, який включає: пари запит-відповідь, обмеження області значень полів та предикати для забезпечення логічної сумісності компонентів. Використання даного контракту дозволяє не лише перевіряти сумісність API компонент веб-серверів, але і 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 області значень полів та їх типів під час виконання тестів дозволить надати розробнику на етапі тестування повноту інформації про можливі проблеми під час роботи двох систем. Оскільки онтологія повинна постійно підтримуватися експертами в галузі, то використання розширених контрактних тестів дозволить зменшити імовірність помилок внаслідок 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 компонент, контракте тестування, DevOps.

Вступ

Згідно із статистикою використання Agile та DevOps методологій розподіляється як 47 % та 37 % відповідно. Зазначимо також, що ці методології можуть об’єднуватися в межах одного проекту 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 впродовж усього циклу розробки та може бути переданий (розгорнутий) для кінцевих користувачів в будь-який момент. Однією із складових даних процесів є автоматизовані тести, що

зменшують кількість роботи, яку повинна виконувати команда тестувальників перед кожним релізом. З 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 у релізах та необхідність термінового розгортання виправлень (hotfix), що може збільшити терміни виконання проекту.

Згідно із дослідженнями компанії Postman, Inc у 2023 році (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 може бути складно відстежити, особливо, коли вони стосуються внутрішньої функціональності.

Постановка проблеми

Основною проблемою є реагування на зміни у бізнес-логіці системи-постачальника. Зазвичай, великим релізам передують кількаразові 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 область визначення полів запиту та відповіді, що може призвести до помилок у роботі систем, коли дані вводяться користувачем.

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 реалізовуються.

Станом на зараз, де-факто стандартом розробки програмних продуктів є Agile або DevOps методології. Оскільки вони часто змішуються, то буває складно їх розділити і чітко визначити якої саме методології дотримується при 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 реалізовують DevOps методологію та включають у себе: безперервну інтеграцію, безперервну доставку та моніторинг. Поетапний DevOps процес зображений на рисунку 1. Нечіткість у визначені меж між цими методологіями часто стає причиною, що проект визначають як “Agile з DevOps практиками”. Насправді для перевірки компонент на сумісність не має значення її місце у методології проекту, важливими є наявність тестів та регулярність їх виконання. Таким чином, не зважаючи на обрану методологію проекту, єдиним місцем для реалізації автоматизованих тестів перевірки інтеграції між системами є CI/CD процес.

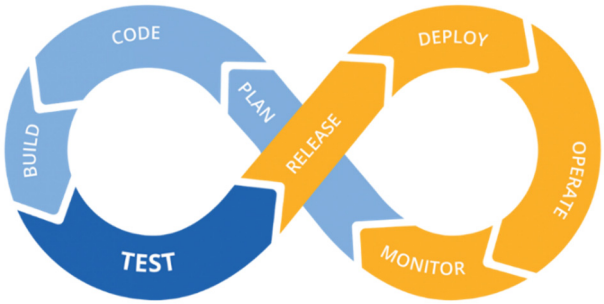

Рис. 1. DevOps процес

Фактично кожний проект, що є достатньо складним, має побудований той чи інший 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 (часто, Pipeline) – це набір плагінів, що дозволяє реалізувати CI/CD процеси проекту 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 рисунку 2.

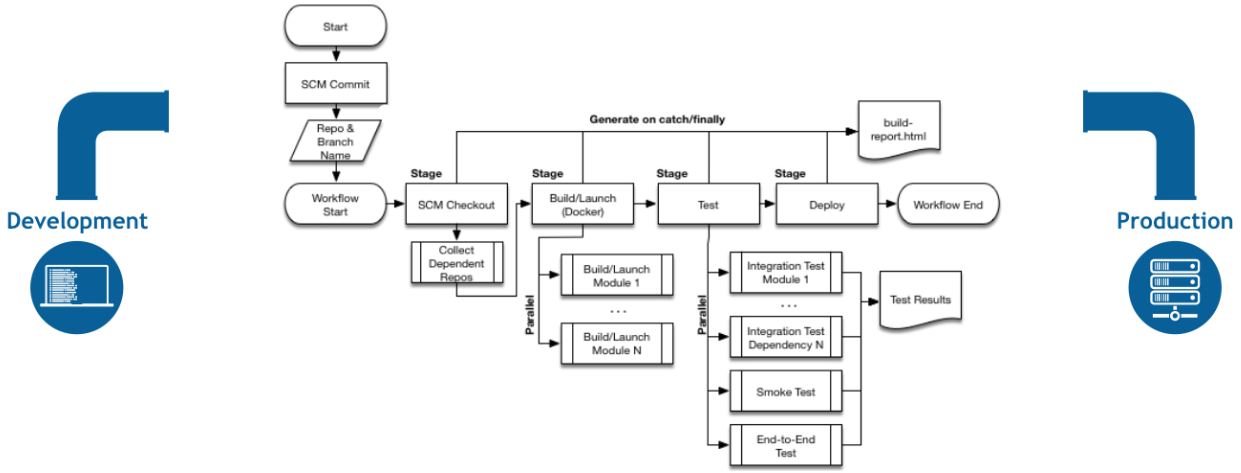

Рис. 2. Загальний вигляд Jenkins Pipeline

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. Крім того, вони можуть використовуватися як контрольна точка якості. Кожний CI/CD процес будується під конкретний проект, враховуючи його вимоги, особливості, бюджет, план випуску версій, архітектуру та інші характеристики.

На даний момент не існує інструменту призначеного для валідації онтологій двох різних систем або для перевірки відповідності онтології структурі однією з систем. Існують такі інструменти як 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 використовується OntoQA, що має змогу оцінити зв’язність, сумісність та повноту онтології. Цей інструмент вперше був описаний у 2005 році, а станом на зараз проект давно не отримував оновлень та може бути несумісний із сучасними засобами розробки [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 веб-серверів та врахування обмежень допустимих значень полів.

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

Виклад основного матеріалу

Оскільки більшість проектів 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 даний 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, в той час як внутрішня команда займається розробкою та підтримкою тестів. Крім того, команді розробників системи-постачальника необхідно розглянути наступні сценарії, щоб тести працювали стабільно:

1. паралельне виконання тестів;
2. відсутність файлу, що містить онтологію;
3. додаткові канали комунікації;
4. запуск тестів не лише на вимогу, а і як заплановане завдання (cron-завдання).

На рисунку 3 зображено діаграму послідовності вищенаведеного процесу тестування. Її загальний вигляд включає 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). Рекомендується використовувати кілька каналів, як і для високої надійності, так і розділяти їх для різних команд або в залежності від рівня важливості повідомлення.

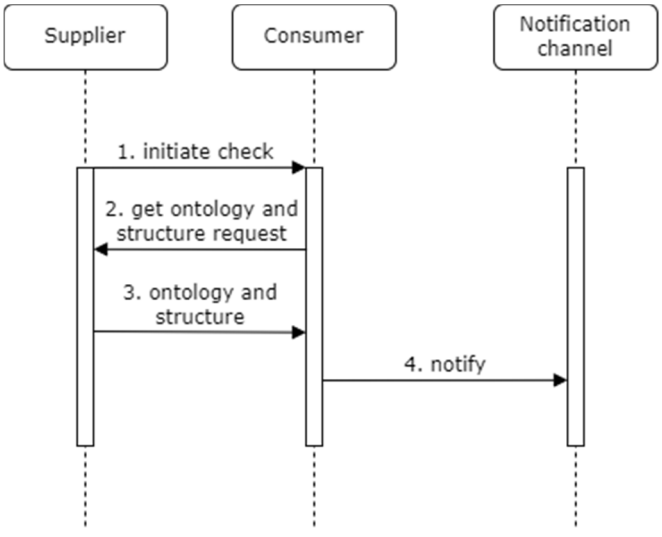

Рис. 3. Діаграма послідовності

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 запустити тести, а і 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 випадку використання статистики, вона повинна зберігатися у базі даних (актор “Database”) та бути доступною виключно для команди розробників.

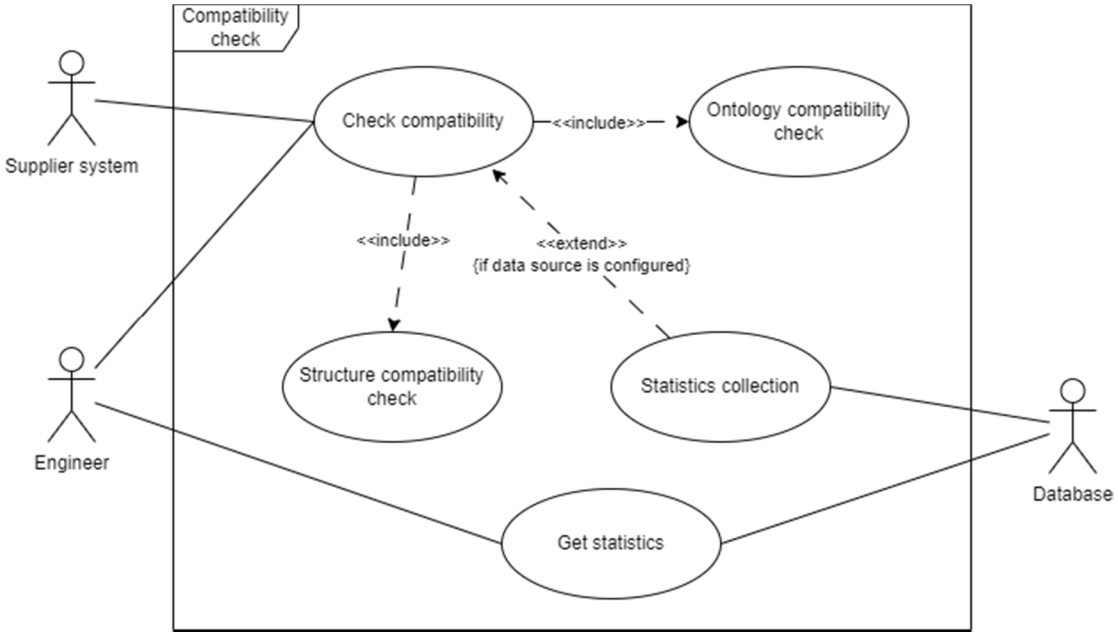

Рис. 4. Діаграма прецедентів

Власне процес перевірки сумісності компонент зображений 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 в залежності від вимог – це результат виконання перевірки чи усі правила існують.

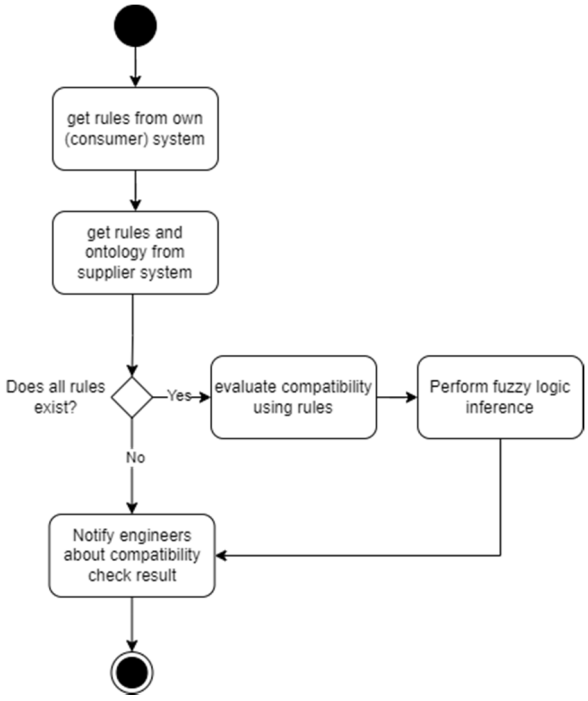

Рис. 5. Діаграма діяльності

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

1. використання попередньої онтології (можливо за умови, що онтологія була завантажена хоча б раз);
2. термінова комунікація із службою підтримки системи-постачальника;
3. пропуск тестів перевірки сумісності без урахування їх у результаті виконання усього пакету тестів.

Для правильного та повного виконання тестів було розширено стандартний контракт та додано до нього перевірку на область 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 дозволяє 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

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)
Requst-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 правили 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 (представляється 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) та DTO зі сторони системи-споживача та концепт і DTO зі сторони 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 усі 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 співпадають **або** кількість обов'язкових полів не співпадає
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 підключення до бази даних то
Зберегти звіт у базі даних
Кінець якщо

Результат виконання тестів може бути використаний для перевірки контрольних точок якості і згідно із 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 даного дослідження є аналіз місця виконання автоматизованих контрактних тестів для перевірки сумісності окремих компонент веб-серверів у структурі Agile/DevOps процесів та розробка розширених контрактів для перевірки структури, типів, обмежень та логічного зв'язку між компонентами веб-серверів. Оскільки, над різними складовими комплексної системи часто працюють різні команди, 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. Даний підхід був продемонстрований діаграмами, що описують процес виклику тестів та процес їх виконання. Крім того, був наведений псевдокод алгоритму перевірки систем на сумісність з використанням нечіткого логічного виведення для надання рекомендацій розробникам та керівникам проектів. Перспективою подальших досліджень є розробка програмних модулів та інтерфейсів для реалізації системи перевірки сумісності окремих компонент веб-серверів.

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. (n.d.). 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). (2023, December

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). *jFuzzyLogic: 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. (n.d.). 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*). jFuzzyLogic: 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 Anal*ysis. *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

Krupa Dmytro

Lviv Polytechnic National University,
Department of Information Systems and Networks, Lviv, Ukraine

E-mail: dmytro.v.krupa@lpnu.ua, ORCID: 0009-0002-6087-9988

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.