Міністерство освіти і науки України
Сумський державний університет

Бойко О. В.

ІНТЕГРОВАНІ ІНФОРМАЦІЙНІ СИСТЕМИ

Конспект лекцій

Суми
Сумський державний університет
2023

Міністерство освіти і науки України
Сумський державний університет

ІНТЕГРОВАНІ ІНФОРМАЦІЙНІ СИСТЕМИ

Конспект лекцій
для студентів спеціальності *122 «Комп'ютерні науки»*
освітнього ступеня «магістр»
усіх форм навчання

Затверджено
на засіданні кафедри інформаційних технологій
як конспект лекцій
із дисципліни «Інтегровані інформаційні системи».
Протокол № 2 від 29.08.2023.

Суми
Сумський державний університет
2023

Інтегровані інформаційні системи : конспект лекцій / укладач: О. В. Бойко. – Суми : Сумський державний університет, 2023. – 130 с.

Кафедра інформаційних технологій

ЗМІСТ С.

ВСТУП

У сучасному світі інформаційні системи є невід'ємною частиною будь-якого бізнесу. Вони використовуються для управління ресурсами, оброблення даних, ухвалення рішень та інших важливих завдань.

Інтеграція інформаційних систем – це процес об'єднання двох або більше інформаційних систем для спільного використання даних і ресурсів. Інтегровані інформаційні системи можуть забезпечити низку переваг, таких як підвищення ефективності бізнесу, покращання якості ухвалення рішень і зниження витрат.

Дисципліна «Інтегровані інформаційні системи» надає студентам фундаментальні знання про технології інтеграції інформаційних систем. Студенти дізнаються про різні підходи до інтеграції інформаційних систем, про методи проєктування та реалізації інтегрованих інформаційних систем, а також про інструменти та технології, які використовуються для інтеграції інформаційних систем.

Дисципліна також надає студентам практичні навички, необхідні для проєктування, реалізації та підтримки інтегрованих інформаційних систем. Студенти будуть виконувати практичні завдання та проєкти, які допоможуть їм застосувати одержані знання на практиці.

Навчальний матеріал конспекту лекцій з дисципліни «Інтегровані інформаційні системи» відповідає вимогам освітньої програми спеціальності 122 «Комп'ютерні науки». Конспект лекцій містить теоретичні основи інтеграції інформаційних систем, а також практичні аспекти проєктування, розроблення та впровадження ІІС.

Конспект лекцій розрахований на студентів усіх форм навчання. Матеріали конспекту можуть бути використані для самостійного вивчення дисципліни, а також широким загалом читачів для підвищення загальної обізнаності в галузі інтеграції інформаційних систем.

Тема 1. Технології інтеграції інформаційних систем

Лекція 1. Основні поняття та визначення інтеграції інформаційних систем

Інтеграція інформаційних систем (ІІС) – це процес об'єднання двох або більше інформаційних систем (ІС) для спільного використання даних і функціональних можливостей. Інтеграція ІС дозволяє організаціям підвищити ефективність роботи, покращити якість обслуговування клієнтів і зменшити витрати. До неї належить поєднання апаратних, програмних і мережевих компонентів для створення уніфікованої системи, яка може працювати безперебійно та ефективно.

ІІС може бути складним і витратним завданням. Тому важливо провести ретельне планування та виконання перед початком проєкту. Деякі **основні фактори**, які потрібно враховувати під час планування ІІС, містять:

- Цілі інтеграції: Чому організація хоче інтегрувати свої ІС? На які конкретні переваги вона очікує?
- Досяжність: Чи є технічно можливим інтегрувати існуючі ІС?
- Вартість: Скільки коштуватиме проєкт інтеграції?
- Час: Скільки часу займе проєкт?
- Ризик: Які ризики пов'язані з проєктом інтеграції?

Після того як план інтеграції розроблений, організація може розпочати реалізацію проєкту.

Основні **етапи реалізації проєкту** містять:

- аналіз існуючих ІС: оцінювання поточної ситуації та визначення потреб у змінах;
- розроблення архітектури інтеграції: визначення того, як ІС будуть інтегровані;
- упровадження змін: імплементація змін в ІС;
- тестування інтеграції: перевірка того, що ІС інтегровані належним чином;
- підтримка інтеграції: надання підтримки користувачам після впровадження інтеграції.

Існує кілька **типів ІІС**, зокрема:

1. Вертикальна інтеграція: цей тип інтеграції містить інтеграцію різних компонентів системи, таких як апаратне та програмне забезпечення, в межах однієї організації чи окремої галузі.

2. Горизонтальна інтеграція: цей тип інтеграції передбачає інтеграцію різних систем, які виконують подібні функції, але належать до різних організацій або галузей.

3. Інтеграція корпоративних додатків (EAI): цей тип інтеграції містить інтеграцію різних додатків, які підтримують різні бізнес-функції в організації, такі як бухгалтерський облік, управління запасами, управління взаємовідносинами з клієнтами та управління ланцюгом поставок.

4. Інтеграція даних: цей тип інтеграції передбачає об'єднання даних із різних систем або джерел для забезпечення єдиного перегляду даних, що може покращити процес ухвалення рішень і бізнес-процеси.

5. Хмарна інтеграція: цей тип інтеграції передбачає під'єднання різних хмарних додатків і систем для забезпечення їхньої бездоганної взаємодії.

6. Інтеграція бізнес-бізнес (B2B): цей тип інтеграції передбачає інтеграцію систем між двома або більше організаціями для оптимізації зв'язку, обміну даними та бізнес-процесів.

7. Інтеграція Інтернету речей (IoT): цей тип інтеграції передбачає під'єднання різних пристроїв, датчиків і систем для забезпечення зв'язку та обміну даними в міжмашинному (M2M) середовищі.

Кожен тип системної інтеграції має різні цілі й може надавати різні переваги залежно від потреб і цілей організації.

Вертикальна інтеграція – це стратегія, яка передбачає об'єднання різних етапів виробничого процесу в одну організацію. Це може бути зроблено за допомогою придбання або створення нового бізнесу.

Вертикальна інтеграція може бути зворотною або прямою. Зворотна інтеграція передбачає придбання компанією постачальників сировини або інших ресурсів. Пряма інтеграція передбачає придбання компанією роздрібних продавців або інших каналів збуту.

Вертикальна інтеграція може мати низку переваг, таких як:

- Більший контроль над ланцюгом постачання. Компанії, які вертикально інтегровані, мають більше контролю над своїми поставками, що може допомогти їм уникнути збоїв у постачаннях і забезпечити стабільність цін.
- Зменшення залежності від сторонніх постачальників. Вертикальна інтеграція може допомогти компаніям зменшити свою залежність від сторонніх постачальників, що може дати їм більше переговорної сили й зменшити ризик перебоїв у постачаннях.
- Підвищення ефективності та економія коштів. Вертикальна інтеграція може допомогти компаніям підвищити ефективність своїх операцій і заощадити кошти, наприклад, шляхом уникнення витрат на логістику та комісій за посередництво.

Проте вертикальна інтеграція також може мати низку недоліків, таких як:

- Підвищення ризику. Вертикально інтегровані компанії можуть мати більший ризик, оскільки вони залучені до більшої кількості видів діяльності.
- Зниження гнучкості. Вертикально інтегровані компанії можуть мати меншу гнучкість, ніж компанії, які не є вертикально інтегрованими, оскільки вони можуть бути обмежені своїми власними ресурсами та можливостями.

Приклади вертикальної інтеграції:

- Ford Motor Company: на початку XX століття компанія Ford вертикально інтегрувала більшу частину свого ланцюжка поставок, володіючи та керуючи металургійними

заводами, каучуковими плантаціями та іншими ресурсами, необхідними для виробництва своїх автомобілів.

- Amazon: Amazon реалізував як зворотну, так і пряму вертикальну інтеграцію. Компанія придбала склади, компанії з транспортування та доставки, і навіть виробляє власні продукти, такі як Amazon Echo.
- Coca-Cola: Coca-Cola здійснила вертикальну інтеграцію шляхом придбання розливних компаній для контролю над виробництвом і поширенням своєї продукції.
- Apple: Apple вертикально інтегрована шляхом придбання компаній, які постачають апаратні компоненти для їхніх продуктів, таких як мікрочипи та дисплеї, щоб забезпечити якість і стабільність поставок.
- Marriott International: компанія Marriott реалізувала вертикальну інтеграцію, володіючи та керуючи багатьма готелями, якими вона керує, а також системами бронювання та програмами лояльності.

Горизонтальна інтеграція – це стратегія бізнесу, яка передбачає об'єднання двох або більше компаній, які працюють в однакових або схожих галузях. Мета горизонтальної інтеграції – збільшити частку ринку, зменшити конкуренцію та підвищити ефективність.

Горизонтальна інтеграція може набувати різних форм, наприклад:

- Злиття: дві або більше компаній об'єднують свою діяльність для створення нової організації.
- Поглинання: одна компанія купує іншу компанію та інтегрує її діяльність у свою власну.

Горизонтальна інтеграція може надати компанії такі переваги:

- Економія за рахунок масштабу: об'єднання двох або більше компаній може призвести до зниження витрат на виробництво та маркетинг.

- Посилення ринкової влади: більша компанія може мати більшу ринкову владу, що може дозволити їй встановлювати ціни та умови продажу.
- Підвищення ефективності: об’єднання двох або більше компаній може призвести до більш ефективного використання ресурсів.

Проте горизонтальна інтеграція може також створити такі проблеми:

- Необхідність управління більшою організацією: об’єднання двох або більше компаній може призвести до створення більшої організації, що може ускладнити управління та контроль.
- Ризик антимонопольного регулювання: горизонтальна інтеграція може привести до зменшення конкуренції, і відбудеться антимонопольне регулювання.

Приклади горизонтальної інтеграції

- Придбання Disney 21st Century Fox: у 2019 році Disney придбала розважальні активи 21st Century Fox, серед яких були кіно- та телестудії Fox, а також такі популярні франшизи, як «Люди Ікс» і «Сімпсони». Це дозволило Disney збільшити свою частку ринку в розважальній галузі та одержати доступ до нової аудиторії.
- Злиття ExxonMobil: у 1999 році Exxon і Mobil, дві великі нафтові компанії, об’єдналися, щоб створити ExxonMobil, одну з найбільших нафтових компаній у світі. Це дозволило ExxonMobil збільшити свою частку ринку в нафтовій галузі та зменшити конкуренцію.
- Придбання Facebook компанією Instagram: у 2012 році Facebook придбала Instagram, додаток для обміну фотографіями. Це дозволило Facebook розширити свої пропозиції в соціальних мережах і одержати доступ до нової бази користувачів.
- Придбання Yahoo компанією Verizon: у 2017 році Verizon придбала інтернет-бізнес Yahoo, враховуючи службу

електронної пошти, пошукову систему та контент-сайти. Це дозволило Verizon розширити свої цифрові медіа-пропозиції та конкурувати з іншими великими технологічними компаніями, такими як Google і Facebook.

Інтеграція корпоративних додатків (EAI) – це процес об'єднання різних бізнес-додатків в організації, щоб забезпечити безперебійний обмін інформацією та координацію бізнес-процесів.

EAI зазвичай передбачає використання проміжного програмного забезпечення, яке є рівнем програмного забезпечення, що розміщується між різними програмами та дозволяє їм спілкуватися один з одним. Проміжне програмне забезпечення може набувати різних форм, таких як службова шина підприємства (ESB), системи обміну повідомленнями або інтерфейси прикладного програмування (API).

Перевагами EAI є: покращання ефективності та продуктивності; зменшення надмірності даних; покращання процесу ухвалення рішень; краще обслуговування клієнтів; підвищення конкурентоспроможності.

Приклади EAI:

- Інтеграція управління взаємовідносинами з клієнтами (CRM) з іншими бізнес-додатками, такими як автоматизація продажів і маркетингу, може забезпечити повніше уявлення про взаємодію з клієнтами та покращити обслуговування клієнтів.
- Інтеграція управління ланцюгом поставок (SCM) з іншими бізнес-додатками, такими як бухгалтерський облік і управління запасами, може забезпечити кращу координацію виробництва, доставки та управління запасами.
- Інтеграція відділу людських ресурсів (HR) з іншими бізнес-додатками, такими як управління заробітною платою та виплатами, може забезпечити більш ефективне керування даними про співробітників і оброблення заробітної плати.
- Інтеграція Business Intelligence (BI) з іншими бізнес-додатками, такими як продажі та фінансові системи, може

забезпечити кращий доступ до інформації та більш обґрунтоване ухвалення рішень.

Інтеграція даних – це процес об'єднання даних із різних джерел в єдине узгоджене подання. Це дозволяє організаціям краще розуміти свої дані та використовувати їх для ухвалення рішень, підвищення ефективності та зниження витрат.

Інтеграція даних може бути складною через складність джерел даних, форматів і систем. Проте це критично важливий компонент сучасних організацій, що керуються даними.

Основні **форми інтеграції даних**:

- Видобуток, перетворення, завантаження (ETL) – це загальний підхід до інтеграції даних, який передбачає вилучення даних із вихідних систем, перетворення їх у загальний формат і завантаження в цільову систему.
- Інтеграція корпоративних додатків (EAI) – це процес об'єднання даних із різних додатків і систем у межах організації.
- Об'єднання даних – це процес доступу до даних із різних джерел у реальному або майже реальному часі без необхідності витягувати, перетворювати та завантажувати дані в цільову систему.
- Реплікація даних – це процес копіювання даних з однієї системи в іншу для підтримання синхронізації даних у різних системах.
- Віртуалізація даних – це процес створення віртуального перегляду даних, які зберігаються в різних джерелах і системах, без необхідності фізичного переміщення даних.

Приклади інтеграції даних:

1. Інтеграція даних про клієнта: інтеграція даних про клієнта з різних систем, таких як продажі, маркетинг і обслуговування клієнтів, може забезпечити більш повне уявлення про клієнта та забезпечити кращу персоналізацію та націлювання маркетингових кампаній.

2. Інтеграція фінансових даних: інтеграція фінансових даних із різних систем, таких як бухгалтерський облік, виставлення рахунків і оброблення платежів, може забезпечити більш точне та своєчасне уявлення про фінансові показники та уможливити кращі фінансові прогнози та складання бюджету.

3. Інтеграція медичних даних: інтеграція медичних даних із різних систем, таких як електронні записи про стан здоров'я (EHR), медичні зображення та пристрої моніторингу пацієнтів, може забезпечити кращу координацію медичної допомоги та покращити результати пацієнтів.

4. Інтеграція даних ланцюга поставок: інтеграція даних ланцюга постачання з різних систем, таких як управління запасами, логістика та закупівлі, може забезпечити кращу видимість операцій ланцюга постачання та покращити планування та виконання ланцюга постачання.

Хмарна інтеграція – це процес об'єднання хмарних додатків і служб з іншими програмами та службами, як локальними, так і в інших хмарних середовищах. Вона дозволяє організаціям використовувати переваги хмарних систем, одночасно інтегруючи їх з іншими системами.

Хмарна інтеграція може набувати різних форм, наприклад, інтерфейси прикладного програмування (API), системи обміну повідомленнями або керовані подіями архітектури.

Приклади хмарної інтеграції:

- Інтеграція «хмара-хмара»: об'єднання хмарних додатків і служб між собою для безперебійного обміну даними та зв'язку.
- Інтеграція «хмара-локація»: об'єднання хмарних додатків і служб із локальними системами, такими як системи CRM або ERP, для створення гібридних хмарних середовищ і полегшення обміну даними.
- Інтеграція «хмара-хмара-локація»: об'єднання кількох хмарних систем і локальних систем, що забезпечує обмін даними та зв'язок між ними.

- Інтеграція «від хмари до краю»: об’єднання хмарних додатків і служб із периферійними пристроями, такими як пристрої IoT або периферійні шлюзи, для забезпечення оброблення та аналізу даних у реальному часі на межі.

Інтеграція B2B – це обмін даними та бізнес-процесами між двома або більше підприємствами. Це дозволяє автоматизувати транзакції та обмін даними, покращувати видимість ланцюга поставок, скорочувати ручне оброблення та зменшувати витрати.

Приклади інтеграції B2B:

- Інтеграція ланцюжка поставок: інтеграція бізнес-процесів і даних між постачальниками, виробниками, дистриб’юторами та роздрібними торговцями.
- Інтеграція торгових партнерів: інтеграція бізнес-процесів і даних між торговими партнерами, такими як клієнти або постачальники.
- Фінансова інтеграція: інтеграція фінансових процесів і даних між компаніями, такими як виставлення рахунків, платежі та збори.
- Логістична інтеграція: інтеграція логістичних процесів і даних між підприємствами, такими як доставка, транспортування та складування.

Інтеграція Інтернету речей (IoT) – це процес об’єднання пристроїв і даних IoT з іншими системами та програмами. Це дозволяє підприємствам збирати, обробляти та аналізувати дані з пристроїв IoT, а також керувати цими пристроями з інших систем.

Інтеграція IoT може бути складною, але вона є важливим компонентом цифрової трансформації та інновацій. Вона дозволяє підприємствам використовувати переваги технологій і програм IoT, таких як покращена операційна ефективність, покращений досвід роботи з клієнтами та краще ухвалення рішень на основі даних у реальному часі.

Ось кілька прикладів інтеграції IoT:

- Інтеграція промислового Інтернету речей (IIoT): інтеграція пристроїв і даних IoT із промисловими системами керування, такими як системи диспетчерського контролю та збирання даних (SCADA) або системи планування ресурсів підприємства (ERP), для підвищення ефективності роботи та управління активами.
- Інтеграція з розумним будинком: інтеграція пристроїв і даних IoT із системами розумного будинку, такими як голосові помічники або мобільні програми, для забезпечення контролю та моніторингу домашніх пристроїв і приладів.
- Інтеграція з підключеним автомобілем: інтеграція пристроїв і даних IoT з підключеними автомобілями та телематичними системами, такими як GPS або датчики для відстеження та моніторингу роботи та використання автомобіля в реальному часі.
- Інтеграція Інтернету речей у сфері охорони здоров'я: інтеграція пристроїв і даних IoT із системами охорони здоров'я, такими як електронні записи про стан здоров'я (EHR) або платформи телемедицини, для покращання результатів лікування пацієнтів і віддаленого моніторингу.

Лекція 2. Підходи до інтеграції інформаційних систем

Під терміном **«інтеграція»** можна розуміти об'єднання ІС, застосунків, різних компаній або людей. Виділяють зовнішню та внутрішню інтеграцію: внутрішня передбачає об'єднання різних корпоративних застосувань в одній організації, зовнішня – об'єднання ІС різних організацій.

Немає універсального інтеграційного рішення, яке могло б підійти всім. Кожне інтеграційне рішення є унікальним і вимагає застосування різних підходів до об'єднання додатків. Однак, незважаючи на різноманітність завдань інтеграції та способів зв'язування додатків, різними авторами робилися спроби виділити низку базових класів інтеграційних рішень, використовуючи мову шаблонів.

Шаблон виконує роль референтної моделі та є абстрактним описом типової задачі й способів її вирішення.

Існують такі основні типові інтеграційні підходи: інтеграція на рівні даних; інтеграція на рівні бізнес-функцій і бізнес-об'єктів; інтеграція на рівні бізнес-процесів; портали».

Різні шаблони інтеграції відповідають відповідним цілям (рис. 1.1).

Шаблони інтеграції		
Ціль інтеграції		**Шаблони архітектурного проміжного шару**
Єдине подання корпоративних даних		Агрегація сутностей
Розподілений бізнес-процес		Інтеграція процесів
Єдина точка доступу до даних з декількох джерел		Інтеграція порталів
Рівень інтеграції додатків		**Шаблони інтеграції додатків**
База даних		Інтеграція даних
Рівень бізнес-логіки		Функціональна інтеграція
Рівень користувацького інтерфейсу		Презентаційна інтеграція
Спосіб взаємодії додатків		**Шаблони топології інтеграційного рішення**
Взаємодія «Один-до-одного»		Точка-точка
Взаємодія через центральний «комутатор»		Брокер повідомлень
Відкрита взаємодія		Шина повідомлень
Взаємодія «один-до-багатьох»		Публікація-підписка

Рисунок 1.1 – Шаблони інтеграції

1. **Інтеграція на рівні даних** передбачає наявність у системах баз даних, для роботи з якими необхідно розробити єдиний програмний інтерфейс. До основних технологічних рішень цього підходу відносять: системи реплікації даних; обласні бази даних; використання API для доступу до EPR-систем.

Реплікація є процесом синхронізації даних між різними джерелами. Необхідність у цьому виникає в момент зміни блоку інформації в розподілених системах зберігання, щоб гарантувати коректність і несуперечність даних,

використовуваних в усіх модулях або додатках інформаційної системи. Зазвичай функції реплікації покладають на проміжне ПЗ.

Обласні (федеративні) бази даних надають єдиний інтерфейс до розподілених даних. Це забезпечує інтеграцію множини автономних даних, які можуть бути фізично розміщені на різних пристроях у мережі. Такі бази даних прийнято називати віртуальними.

Використання API для доступу до ERP-систем покликано спростити механізми обміну інформацією між призначеними для користувача застосунками й програмним забезпеченням, призначеним для управління функціонуванням виробничих ІС (ERP).

2. **Інтеграція на рівні бізнес-функцій і бізнес-об'єктів** передбачає реалізацію спільно використовуваних служб (сервісів). Служба може бути набором функцій, який використовується в декількох застосунках. Цей набір служб і буде бізнес-функціями. Під час використання сервісно-орієнтованої архітектури бізнес-функції можна розглядати як бізнес-сервіси, а за компонентного підходу – як бізнес-об'єкти (бізнес-компоненти).

3. **Інтеграція на рівні бізнес-процесів** розрізняється залежно від рівня інтеграції. За внутрішньої інтеграції взаємодіє велика кількість сервісів, а за зовнішньої інтеграції – в основному два. Бізнес-процеси функціонують над виділеними службами, для управління якими існує спеціальна мова, що інтерпретується.

4. **Портали** можна вважати графічними інтерфейсами бізнес-процесів, оскільки вони призначені для персоніфікованого доступу до інформації та консолідації даних із декількох джерел.

Основним призначенням процесу інтеграції є об'єднання функцій додатків або модулів для надання нової функціональності.

Під час інтеграції додатків можна виділити *два основні типи завдань інтеграції*:

1) корпоративних застосунків;

2) додатків із різних ІС.

Для вирішення завдань першого типу застосовують системи EAI, які іноді називаються A2A (Application – to – Application Integration), а для вирішення завдань другого типу застосовуються системи B2B (Business – to – Business Integration).

У деяких випадках складно визначити різницю між інтеграцією A2A і B2B, оскільки складність деяких рішень усередині ІС може перевищувати складність рішень для їх спільного функціонування.

Існують три альтернативні **топології інтеграції**:

1) точка-точка (Point – to – Point);
2) шлюз (hub – and – spoke);
3) шина (Bus).

У топології «точка-точка» всі об'єкти мають прямі зв'язки один з одним (рис. 1.2 а). Кожен зв'язок можна реалізувати будь-яким способом. Варіанти реалізації залежать від вимог і характеристик взаємодії між об'єктами. До недоліків топології можна віднести такі характеристики: недостатня гнучкість; складність підтримки численних з'єднань «точка-точка»; зміни одного об'єкта впливають на об'єкти, що залишилися; логіка маршрутизації часто програмується в коді об'єктів; відсутність загальної моделі безпеки; використання різних API; низька надійність; складність створення фреймворків і підтримки асинхронної взаємодії.

Для скорочення числа використовуваних інтерфейсів необхідно використати топологію із загальним шлюзом (рис. 1.2 б) або топологію із загальною шиною (рис. 1.2 в). Такі моделі інтеграції реалізуються на рівні проміжного ПЗ.

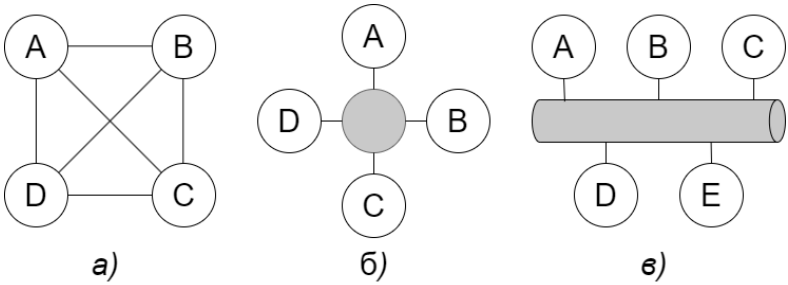

Рисунок 1.2 – Топології інтеграції

Наступним кроком у розробленні інтеграційної архітектури можна вважати появу корпоративної сервісної шини (**Enterprise Service Bus – ESB**).

Деякі автори розглядають систему ESB як наступний ступінь розвитку EAI. Проте є такі відмінності: *EAI – централізована архітектура з обміном інформації через хаб (брокер*), а ESB – шинна архітектура, яка може бути реалізована у вигляді декількох розподілених систем; вона орієнтована на використання відкритих стандартів. Ці дві відмінності демонструють можливість використання ESB як інтеграційної платформи, що дозволяє використати різні механізми.

ESB дозволяє проводити як внутрішню, так і зовнішню інтеграції, і є шиною, працюючою як слабко-зв'язна система, керована подіями. Концепції сервісно-орієнтованої архітектури (СОА) і ESB сильно пов'язані. ESB підтримує принцип реалізації СОА: розділення служби подання та її реалізації.

Функції ESB: надання інтерфейсів взаємодії; відправлення та маршрутизація повідомлень; перетворення даних; реакція на події; управління політиками; віртуалізація. На підставі функцій ESB можна сформувати типовий список вимог, які ставляться користувачеві: велика пропускна спроможність; підтримка декількох стилів інтеграції; забезпечення можливості додаткам працювати з сервісами як безпосередньо, так і через адаптери.

ESB є, по суті, логічним компонентом архітектури, що приводить інтеграційну інфраструктуру у відповідність принципу СОА. Архітектурою, побудованою за принципом ESB, складніше управляти, але вона гнучкіша й масштабована (впровадження СОА не потребує змін в усіх елементах системи, внаслідок чого зможе відбуватися поетапно).

Можна подати ESB у вигляді п'ятирівневої структури:

1. Рівень сполучення покликаний вирішувати проблему використання різних інтерфейсів На цьому рівні функціонують адаптери, які відстежують події в додатках і в інтеграційній підсистемі, і забезпечують перетворення передаваних об'єктів під час взаємодії з транспортною підсистемою. Окрім заздалегідь створених адаптерів інтеграційної платформи існує можливість використати адаптери, створені самостійно. Адаптери можна розділити на дві категорії: 1) технологічні (застосовуються для інтеграції технологічних компонент, за відсутності у них API); 2) адаптери для додатків (застосовуються для інтеграції з конкретним застосунком).

2. Транспортна підсистема надає можливість асинхронної взаємодії інтегрованим застосункам. Цей рівень відповідає також за управління й безпеку інформації, може виконувати маршрутизацію повідомлень та їх оброблення.

3. **Рівень реалізації бізнес-логіки** надає функції для трансформації й маршрутизації повідомлень. На цьому рівні функціонують брокери повідомлень, які обмінюються повідомленнями через транспортну підсистему.

Брокер повідомлень може виконувати такі функції:

1. Прийняття повідомлень та їх відправлення за зазначеними адресами.
2. Перетворення форматів повідомлень.
3. Агрегація та фрагментація повідомлень.
4. Взаємодія з репозиторіями.
5. Вибірка даних через виклики Web-служб.
6. Оброблення помилок і подій.

7. Маршрутизація повідомлень за адресою, змістом, темою.

4. **Управління бізнес-процесами** на одноіменному рівні здійснюється за допомогою мови управління бізнес-процесами (Business Process Execution Language) на основі Web-сервісів.

5. **Рівень бізнес-управління** є надбудовою над попереднім рівнем і призначений для управління бізнес-процесами в термінах відповідної предметної сфери. Підхід ESB має низку переваг і дозволяє будувати інтеграційну архітектуру будь-якої складності. Типова структура інтеграційної системи наведена на рисунку 1.3.

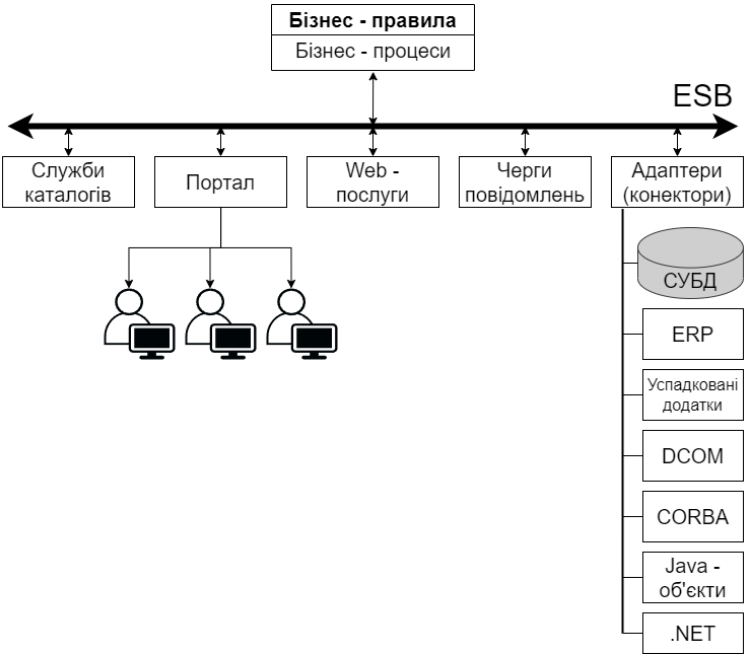

Рисунок 1.3 – Структура інтеграційної системи корпоративної сервісної шини

Тема 2. Формалізація інтегрованих інформаційних систем

Лекція 3. Формалізація інтегрованих інформаційних систем

Формалізація інтегрованих інформаційних систем – це процес подання даних, процесів і взаємозв'язків між складовими ІІС у формалізованому вигляді, що дозволяє використовувати формальні методи для аналізу, проєктування та розроблення ІІС.

Формалізація ІІС має низку переваг, зокрема: покращує розуміння ІІС та її складності; полегшує комунікацію між учасниками проєкту ІІС; зменшує ризик помилок у проєктуванні та реалізації ІІС; покращує якість ІІС.

Формалізація та моделювання ІІС допомагають зрозуміти та описати їх функціональність, структуру, взаємодію та процеси, що відбуваються всередині системи.

Процес формалізації інформаційної системи зазвичай починається з аналізу вимог до системи та визначення її моделі та може бути розбитий на такі етапи:

1. *Визначення вимог до системи*. На цьому етапі визначається функціональність і вимоги до системи. Це може містити аналіз бізнес-процесів, вивчення потреб користувачів та інших зацікавлених сторін.

2. *Аналіз проєктування*. На цьому етапі проводиться аналіз існуючих проєктів, які можуть бути використані як база для проєктування нової системи. Також вивчаються можливості й технології, які можуть бути використані для реалізації системи.

3. *Моделювання системи*. На цьому етапі створюються формальні моделі системи, такі як моделі бізнес-процесів, моделі даних, архітектура системи та інтерфейси. Ці моделі допомагають зрозуміти та описати структуру та функціональність системи.

4. *Розроблення технічного проєкту*. На цьому етапі розробляється технічний проєкт системи, що містить технічні

деталі реалізації, а також опис вимог до обладнання та програмного забезпечення.

5. *Розроблення та впровадження системи*. На цьому етапі проводять розроблення системи, тестування та налаштування. Після успішного тестування систему вводять в експлуатацію.

6. *Підтримання та розвиток системи*. Цей етап передбачає постійну підтримку та розвиток системи з метою забезпечення її надійності та ефективності в роботі.

Формальні методи – це методи, засновані на математичній логіці та теорії автоматів. Вони використовуються для аналізу, проєктування та розроблення ІІС.

До основних видів формальних методів належать:

- *Моделювання*: використання математичних моделей для репрезентування ІІС.
- *Доведення*: використання математичних доказів для перевірки правильності ІІС.
- *Верифікація*: використання формальних методів для перевірки відповідності ІІС її вимогам.
- *Валідація*: використання формальних методів для перевірки відповідності ІІС її цілям.

Моделювання процесу інтеграції є важливим етапом формалізації ІІС. Моделі процесу інтеграції допомагають уявити складні взаємозв'язки між складовими ІІС.

Для моделювання процесу інтеграції можуть використовуватися різні методи, зокрема:

- Сценарій: словесний опис процесу інтеграції.
- Діаграма: графічне подання процесу інтеграції.
- Алгоритм: формальний опис процесу інтеграції.

Побудову моделі системи ми розглядаємо як етап вирішення проблемної ситуації, етап вивчення системи. **Моделлю** називається подання об'єкта, системи чи поняття в деякій абстрактній формі, що є зручною для наукового дослідження. Етапи вивчення інтегрованої системи подані на рисунку 2.1.

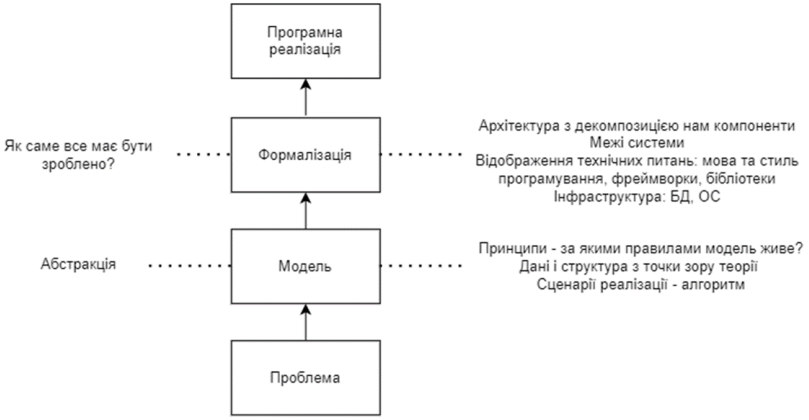

Рисунок 2.1 – Етапи вивчення системи

Етап **моделювання** системи є одним із основних етапів формалізації інтегрованих інформаційних систем. Його **мета** полягає в тому, щоб створити формальну модель системи, яка буде використовуватися в подальшому розробленні та впровадженні системи.

Загалом модель має структуру, зображену на рисунку 2.2. Тут X – множина вхідних змінних системи; Y – множина вихідних змінних системи; P – множина параметрів; F – функція, функціонал, алгоритм або формальне подання залежності змінних Y від змінних X.

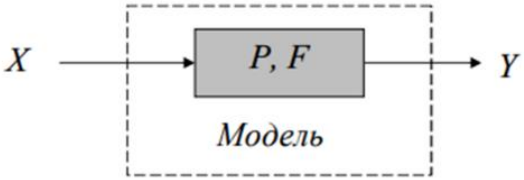

Рисунок 2.2 – Загальний вигляд структури моделі

Кортежний запис моделі

$E: <X,Y,B,A,T,W,F>$,

де X – множина "входів" системи, Y – множина "виходів" системи, B – множина постійних параметрів системи, A – множина змінних параметрів системи, T – множина параметрів процесів в системі, W – оператор динаміки, який дозволяє відобразити множини X, T, B у множину A.

$W(X,T,B) \rightarrow A$,

де F – оператор системи, який дозволяє множини X, T, B, A відобразити у виходи, описує основні функції системи, мету і призначення.

$F(X,T,B,A) \rightarrow Y$.

Кортежні моделі – це моделі, які використовуються для репрезентування даних і зв'язків між ними, що базуються на теорії множин та логіки. У таких моделях дані подані у вигляді кортежів – наборів змінних значень, які містяться у відповідному записі бази даних.

Кортежна модель може бути використана для створення формальних математичних описів даних, що дозволяє використовувати математичні методи та алгоритми для вирішення проблем, пов'язаних із даними. Також кортежні моделі можуть бути використані для репрезентування структури бази даних і для проєктування реляційних баз даних.

Один із прикладів кортежної моделі – модель реляційної бази даних (Relational Database Model). У цій моделі дані подані у вигляді таблиць, де кожен рядок таблиці подає один запис даних, а кожен стовпець подає одне поле даних. Кожен запис у таблиці може бути ідентифікований за допомогою унікального ідентифікатора, який називається ключем.

Кортежна модель може бути більш підходящою для вирішення проблем, пов'заних із даними, **наприклад**:

1. Проєктування баз даних: кортежна модель може бути використана для формального опису взаємозв'язків між таблицями та атрибутами в базі даних, що дозволяє точніше визначати вимоги до даних і забезпечувати їх коректність.

2. Аналіз даних: кортежна модель може бути використана для розуміння взаємозв'язків між різними видами даних та їх властивостями, що дозволяє здійснювати аналіз даних і знаходити залежності між ними.

3. Розроблення програмного забезпечення: кортежна модель може бути використана для формального опису даних та їх взаємозв'язків у програмному коді, що дозволяє зменшити кількість помилок і покращити структуру коду.

4. Моделювання складних систем: кортежна модель може бути використана для моделювання складних систем, що містять велику кількість даних та їх зв'язків, наприклад, системи управління базами даних, мережеві системи та інші.

Моделювання системи може мати такі **підетапи**:

1. *Визначення функціональної моделі системи*. На цьому етапі визначаються основні функції, які повинна виконувати система, та їх взаємодія. Це може містити створення моделі бізнес-процесів, діаграм функцій та ін.

2. *Визначення моделі даних*. На цьому етапі створюється модель даних, яка визначає структуру даних та їх зв'язки в системі. Це може містити створення схем баз даних, діаграм ER (Entity-Relationship) та ін.

3. *Визначення архітектури системи*. На цьому етапі визначається структура та організація компонентів системи, їх взаємодія та інтерфейси між ними. Це може містити створення блок-схем, діаграм компонентів та ін.

4. *Визначення інтерфейсів користувача*. На цьому етапі створюються діаграми, які описують взаємодію користувача з системою, та визначаються основні елементи інтерфейсу користувача, такі як меню, кнопки, поля введення та інші.

5. *Визначення вимог до продуктивності та безпеки системи*. На цьому етапі визначаються вимоги до продуктивності та безпеки системи. Це може містити визначення обсягів даних, кількості користувачів, швидкості роботи системи та ін.

Існує кілька **підходів до моделювання інтегрованих інформаційних систем**. Розглянемо декілька з них:

1. *Модель взаємодії між процесами*: цей підхід базується на моделюванні взаємодії між різними процесами, що взаємодіють у системі. Для цього можуть використовуватися діаграми потоків даних та інші інструменти моделювання процесів.

2. *Функціональна модель*: цей підхід базується на моделюванні функцій, які повинні бути виконані системою. Для цього можуть використовуватись діаграми блоків, діаграми сутностей-з'язків та інші інструменти моделювання функцій.

3. *Модель даних*: цей підхід базується на моделюванні даних, які використовуються системою. Для цього можуть використовуватись діаграми сутностей-зв'язків, схеми баз даних та інші інструменти моделювання даних.

4. *Об'єктно-орієнтована модель*: цей підхід базується на моделюванні об'єктів, які використовуються в системі. Для цього можуть використовуватись діаграми класів, діаграми послідовності та інші інструменти моделювання об'єктів.

Струтурні елементи моделі інтегрованої систем

Загальний вигляд структури моделі інтегрованої системи може варіюватися залежно від конкретної системи та методології моделювання, що використовується. Але здебільшого **модель інтегрованої системи** може бути подана у вигляді таких **структурних елементів**:

1. *Бізнес-процеси (Business Processes).* Бізнес-процеси є основним елементом моделі інтегрованої системи. Вони відображають послідовність дій та операцій, які потрібні для виконання конкретного завдання у системі.

2. *Компоненти системи (System Components).* Компоненти системи можуть мати апаратне та програмне забезпечення, бази даних, мережеві засоби, інтерфейси користувача та інші компоненти, які використовуються в системі.

3. *Дані та інформаційні потоки (Data and Information Flows).* Дані та інформаційні потоки відображають рух даних та інформації між компонентами системи та бізнес-процесами.

4. *Ролі та користувачі (Roles and Users).* Ролі та користувачі відображають різні типи користувачів та їх ролі в системі. Ці ролі та користувачі можуть мати різні рівні доступу та права в системі.

5. *Управління та моніторинг (Management and Monitoring).* Управління та моніторинг відображають процеси управління та моніторингу системи. Ці процеси можуть містити механізми забезпечення безпеки, моніторингу виконання бізнес-процесів і компонентів системи, управління конфігурацією та інші процеси.

Нотації та методи моделювання інтегрованої системи

Існує кілька **нотацій і методів моделювання** функціональних моделей систем, серед яких можна виокремити такі:

1. *Діаграма потоків даних (DFD - Data Flow Diagram).* Це графічна нотація, що використовується для моделювання потоків даних і процесів оброблення цих даних у системі. На діаграмах потоків даних зображуються процеси, що обробляють дані, які передаються між процесами, та зовнішніх сутностей (користувачів, систем, джерел даних тощо), з якими система взаємодіє.

2. *Фнкціональна схема (Function Flow Diagram).* Це нотація, що використовується для моделювання послідовності виконання функцій у системі та взаємодії між ними. На діаграмах функцій зображуються функції, які повинні бути виконані в системі, та зв’язки між ними, що показують порядок виконання функцій s передачу даних між ними.

3. *Діаграма сценаріїв (Use Case Diagram).* Це нотація, що використовується для моделювання взаємодії користувача з системою та сценаріїв використання системи. На діаграмах сценаріїв зображуються користувачі системи, їх дії та сценарії

використання системи, які відображають, як користувачі взаємодіють із системою.

4. *Діаграма станів (State Diagram).* Це нотація, що використовується для моделювання станів.

5. *Ієрархічна модель функцій (Function Hierarchy).* Це метод, що використовується для подання функцій системи в ієрархічній формі, де кожна функція розбивається на більш прості функції, що повторюються в інших частинах системи. Ієрархічна модель функцій дозволяє більш детально проаналізувати роботу системи та знайти шляхи оптимізації її функціонування.

6. *Методологія IDEF (Integrated Definition for Function Modeling).* Це методологія, що використовується для моделювання функцій і процесів у системах. Методологія IDEF містить кілька нотацій, які дозволяють подати функції та їх взаємодії в системі. Крім того, методологія IDEF містить методи аналізу та проєктування систем, що дозволяють ефективно виконувати завдання формалізації та моделювання систем.

7. *Методологія BPMN (Business Process Model and Notation).* Це методологія, що використовується для моделювання бізнес-процесів і взаємодії між різними діловими процесами в організації. Методологія BPMN містить графічні символи та нотації, що дозволяють зображувати бізнес-процеси та їх елементи, такі як рішення, дії, події, стани та інші.

Визначення взаємозв’язків між складовими системи є важливим етапом у розробленні інтегрованих систем. Взаємні зв’язки визначають, як складові системи взаємодіють один з одним, і як вони впливають на поведінку системи.

Для *визначення взаємозв’язків між складовими системи* можна використовувати різні методи, такі як:

- Аналіз вимог. Вимоги до системи можуть містити інформацію про взаємозв’язки між складовими системи.
- Моделювання системи. Моделі систем можуть використовуватися для візуалізації взаємозв’язків між складовими системи.

- Тестування системи. Тестування системи може допомогти виявити потенційні проблеми з взаємозв'язками між складовими системи.

Функціональні вимоги визначають, що система повинна робити. Вони можуть містити такі аспекти, як:

- Функціональність системи. Що система повинна робити?
- Вхідні дані та вихідні дані системи. Які дані система приймає на вході та генерує на виході?
- Поводження системи. Як система поводиться в різних ситуаціях?

Нефункціональні вимоги визначають, як система повинна працювати. Вони можуть містити такі аспекти, як:

- Надійність системи. Як часто система зазнає відмов?
- Безпека системи. Чи є система безпечною для використання?
- Ефективність системи. Як швидко система працює?
- Доступність системи. Як часто система доступна для використання?

Процедури тестування та валідації формалізованих інтегрованих систем повинні бути розроблені для забезпечення того, що система відповідає вимогам і функціонує належним чином. Тестування може містити такі методи, як:

- Тестування функціональних вимог. Тестування функціональних вимог перевіряє, чи система виконує функції, які від неї вимагаються.
- Тестування нефункціональних вимог. Тестування нефункціональних вимог перевіряє, чи система відповідає вимогам щодо надійності, безпеки, ефективності та доступності.
- Тестування системної інтеграції. Тестування системної інтеграції перевіряє, чи складові системи взаємодіють один з одним належним чином.

- Тестування продуктивності. Тестування продуктивності перевіряє, чи система відповідає вимогам щодо продуктивності.
- Тестування безпеки. Тестування безпеки перевіряє, чи система безпечна для використання.

Валідація формалізованих інтегрованих систем повинна бути використана для забезпечення того, що система відповідає своїм цілям. Валідація може містити такі методи, як:

- Аналіз вимог. Аналіз вимог перевіряє, чи система відповідає вимогам, які були визначені для неї.
- Експертний огляд. Експертний огляд перевіряє, чи система відповідає найкращим практикам і стандартам.
- Користувачі. Користувачі системи повинні бути запрошені взяти участь у тестуванні та валідації системи, щоб забезпечити, що система відповідає їхнім потребам.

Тема 3. Архітектура та визначення інформаційних потреб до інтегрованих систем

Лекція 4. Забезпечення якості інтегрованої системи та методи її контролю

Якість інтегрованої системи визначається її здатністю задовольняти потреби користувачів і відповідати поставленим цілям. Основні **критерії якості** інтегрованої системи містять:

- Довірність – здатність системи забезпечувати надійне та безперебійне функціонування.
- Ефективність – здатність системи забезпечувати високу продуктивність та ефективність використання ресурсів.
- Безпека – здатність системи захищати дані від несанкціонованого доступу, використання та зміни.
- Доступність – здатність системи бути доступною користувачам у потрібний час та у потрібних місцях.
- Модульність – здатність системи бути легко адаптованою до змінних вимог.
- Інтегрованість – здатність системи ефективно взаємодіяти з іншими системами та зовнішніми системами.
- Зручність використання – здатність системи бути легкою у використанні та навчанні.
- Доступність інформації – здатність системи надавати користувачам необхідну інформацію в зрозумілій формі.

Окрім цих загальних критеріїв, якість інтегрованої системи може також визначатися її специфічними характеристиками, такими як:

- Точність – здатність системи забезпечувати точність даних і результатів.
- Повнота – здатність системи забезпечувати повний обсяг необхідних даних і результатів.
- Частковість – здатність системи забезпечувати швидке одержання даних і результатів.

- Повнота – здатність системи забезпечувати широкий спектр можливостей для використання.
- Гнучкість – здатність системи адаптуватися до різних сценаріїв використання.
- Економічність – здатність системи забезпечити економічну ефективність.

Існує низка **стандартів** і рекомендацій, які допомагають забезпечити якість інтегрованих систем. Найвідомішими з них є:

- ISO 9001 – міжнародний стандарт системи управління якістю.
- ISO/IEC 27001 – міжнародний стандарт системи управління інформаційною безпекою.
- ISO/IEC 25010 – міжнародний стандарт якості програмного забезпечення.
- TOGAF – методологія архітектури корпоративних додатків.
- ITIL – набір методологій і практик для управління інформаційними технологіями.
- COBIT – фреймворк для управління інформаційними технологіями.

Ці стандарти та рекомендації допомагають розробникам та впровадникам інтегрованих систем забезпечити їхню якість і відповідність вимогам користувачів.

На додаток до цих стандартів і рекомендацій, існує безліч інших **інструментів і методів**, які можуть бути використані **для забезпечення якості** інтегрованих систем. До них належать:

1. Аудит системи. Аудит є одним із найефективніших методів контролю якості інтегрованої системи. Аудитори проводять оцінювання ефективності системи, перевіряючи документацію, виконання процедур і результати роботи.
2. Вимірювання та аналіз результатів. Для забезпечення якості інтегрованої системи необхідно збирати дані та проводити їх аналіз. Це допомагає ідентифікувати проблеми та знайти способи їх вирішення.

3. Внутрішній контроль. Внутрішній контроль – це система процедур і політик, яка розробляється організацією для забезпечення ефективного виконання процесів та оптимізації ризиків. Цей метод містить проведення самоперевірок та оцінювань системи.

4. Зовнішній контроль. Зовнішній контроль – це проведення оцінювання системи з боку незалежних організацій. Цей метод контролю допомагає забезпечити незалежне оцінювання системи та її відповідність стандартам.

5. Тестування системи. Тестування – це процес, який допомагає визначити функціональність системи та її відповідність вимогам. Тестування може бути проведене вручну або автоматично.

6. Управління змінами. Зміни в системі можуть впливати на її ефективність та якість.

У галузі обслуговування клієнтів методи контролю можуть містити:

1. Відстеження клієнтських скарг. Цей метод допомагає збирати та аналізувати скарги клієнтів для визначення проблем і знайти способи їх вирішення.

2. Вимірювання та аналіз задоволеності клієнтів. Цей метод допомагає збирати та аналізувати дані щодо задоволеності клієнтів для визначення проблем і знайти способи їх вирішення.

3. Навчання та підготовка персоналу.

Для забезпечення якості на етапі розроблення ІІС використовують такі **методи контролю якості**:

1. Тестування програмного забезпечення. Цей метод містить виконання різних тестів для перевірки роботи програмного забезпечення, зокрема інтеграційних, системних, функціональних та інших.

2. Оцінювання проєкту. Цей метод містить аналіз різних параметрів проєкту, таких як кількість часу, коштів і ресурсів, які витрачені на розроблення ІІС.

3. Використання стандартів. Для забезпечення якості розробки ІІС використовують різні стандарти, такі як ISO 9001, ISO 27001, ITIL та інші. Вони допомагають забезпечити відповідність розроблення ІІС вимогам і стандартам.

4. Аудит коду. Цей метод містить перевірку якості програмного коду ІІС для виявлення можливих помилок і проблем.

5. Використання методів управління проєктом. Для забезпечення якості розроблення ІІС використовують різні методи управління проєктом, такі як Agile, Scrum, Waterfall та інші.

6. Використання засобів автоматизації. Для забезпечення якості розроблення ІІС використовують різні засоби автоматизації, такі як Continuous Integration (CI), Continuous Delivery (CD) та інші.

Continuous Integration (CI) – це підхід до розроблення програмного забезпечення, який полягає в постійному злитті (інтеграції) змін до вихідного коду з метою швидкого виявлення помилок та забезпечення високої якості продукту.

У методології Continuous Integration розробники здійснюють регулярні зміни в коді та автоматично проводять тестування системи. Це дає можливість швидко виявляти будь-які помилки в коді та вирішувати їх ще до того, як вони стануть серйозними проблемами.

Щоб реалізувати CI, використовують різні інструменти, такі як системи контролю версій (наприклад, Git), автоматичні системи збирання (наприклад, Jenkins, Travis CI), засоби тестування (наприклад, JUnit, PHPUnit) та інші.

Переваги використання Continuous Integration: швидша реакція на зміни в коді; зменшення ризику з'єднання з розробкою; забезпечення високої якості програмного забезпечення; зменшення витрат на тестування та налагодження; зменшення часу, необхідного для випуску нової версії програмного забезпечення.

Недоліки використання Continuous Integration: потребує додаткових зусиль для підтримки; необхідно забезпечувати якісний код, щоб зменшити кількість помилок під час інтеграції; потребує значної автоматизації тестування, що може бути складним і часомірним завданням.

Continuous Delivery (CD) – це підхід до розроблення програмного забезпечення, який полягає в постійному автоматичному випуску (доставці) нових версій продукту.

У методології Continuous Delivery розробники автоматизують процеси, пов'язані з випуском нової версії продукту, враховуючи процеси тестування, збирання, доставки та розгортання. Це дає можливість забезпечувати швидку доставку нових функцій і виправлень до користувачів.

Щоб реалізувати CD, використовують різні інструменти, такі як системи контролю версій, автоматичні системи збирання, засоби автоматичного тестування, системи розгортання та інші.

Переваги використання Continuous Delivery: швидка доставка нових функцій і виправлень; зменшення ризику з'єднання з розробкою; забезпечення високої якості програмного забезпечення; зменшення витрат на тестування та налагодження; зменшення часу, необхідного для випуску нової версії програмного забезпечення; збільшення швидкості відгуку на вимоги користувачів та зміни ринку.

Недоліки використання Continuous Delivery: потребує додаткових зусиль для підтримки; потребує значної автоматизації тестування та розгортання, що може бути складним і часомірним завданням; необхідно забезпечувати якісний код, щоб зменшити кількість помилок під час випуску.

Основна **різниця між CI та CD** полягає в тому, що CI спрямований на забезпечення сталої якості коду та зниження кількості помилок, тоді як CD фокусується на максимальній автоматизації процесу розгортання програмного забезпечення та його випуску.

Хоча CI та CD можуть використовуватись окремо, вони взаємодіють і доповнюють один одного, формуючи цілий підхід до розроблення програмного забезпечення.

Лекція 5. Архітектура інтегрованих інформаційних систем

Архітектура інтегрованих інформаційних систем – це сукупність принципів, підходів і методів, що використовуються для проєктування, реалізації та експлуатації ІІС. Вона визначає структуру, функціональність, взаємодію та взаємодію компонентів ІІС.

Інтегровані інформаційні системи можуть мати різні рівні абстракції, наприклад, можуть бути поділені на різні модулі, які виконують різні функції. Кожен модуль може бути додатково розбитий на підмодулі. Архітектура ІІС повинна враховувати цю ієрархію складових.

Вимоги до архітектури ІІС:

- Можливість інтеграції. Архітектура повинна забезпечувати можливість інтеграції різних інформаційних систем, що використовуються в організації.
- Ефективність. Архітектура повинна бути ефективною з точки зору використання ресурсів і забезпечення якості обслуговування.
- Модульність. Архітектура повинна бути модульною, щоб забезпечити можливість легкого розширення та модифікації системи.
- Мобільність. Архітектура повинна бути мобільною, щоб забезпечити можливість використання системи на різних платформах і пристроях.
- Безпека. Архітектура повинна забезпечувати безпеку даних і системи.

Основні підходи до побудови архітектури ІІС

Функціональний підхід зосереджується на функціях, які повинна виконувати система. Кожна функція розглядається

окремо, а потім вони об'єднуються в більш складні функціональні блоки.

Компонентний підхід розглядає систему як набір компонентів, які мають визначену функціональну роль та інтерфейс для взаємодії з іншими компонентами. Компоненти можуть бути розроблені як окремі програмні модулі або навіть окремі сервіси.

Об'єктно-орієнтований підхід розглядає систему як набір об'єктів, які взаємодіють між собою. Об'єкти можуть бути реалізовані як окремі програмні модулі або навіть окремі сервіси.

Системний підхід розглядає систему як єдине ціле, що складається з різних компонентів. Система взаємодіє з зовнішнім середовищем і впливає на неї.

Приклади

- *Функціональний підхід:* система управління проєктами, де кожна функція (наприклад, планування ресурсів, контроль витрат тощо) розглядається окремо та потім об'єднується в більш складні функціональні блоки.
- *Компонентний підхід:* веб-додаток для онлайн-магазину, де кожен компонент (наприклад, база даних, система оплати, функціонал корзини тощо) розробляється окремо та має визначену функціональну роль та інтерфейс для взаємодії з іншими компонентами.
- *Об'єктно-орієнтований підхід:* програмне забезпечення для автоматизації бухгалтерського обліку, де окремі об'єкти (наприклад, клієнти, поставки, рахунки тощо) розробляються як окремі програмні модулі та взаємодіють через свої методи та інтерфейси.
- *Системний підхід:* система управління виробництвом, де кожен елемент виробничої ланки розглядається як складова частина системи, що взаємодіє з іншими елементами та впливає на них. Така система містить інформаційну систему, яка забезпечує збирання та оброблення даних про виробничі процеси.

Опис архітектурних рішень – це документ, який містить інформацію про архітектурні рішення, прийняті під час розроблення інформаційної системи. Опис архітектурних рішень може містити:

- Вибір технологій: мови програмування, бази даних, інтерфейси користувача тощо.
- Архітектурні паттерни: типові архітектурні рішення для вирішення специфічних проблем.
- Розподілений дизайн: організація компонентів системи та їх взаємодії в розподіленому середовищі.
- Масштабування: можливості масштабування системи в майбутньому.

Рекомендації щодо реалізації та підтримки архітектурних рішень у майбутньому можуть містити:

- Рекомендації щодо розвитку системи: можливі напрямки розвитку системи та варіанти додавання нових функціональних можливостей.
- Рекомендації щодо технічної підтримки: процедури технічної підтримки та розроблення системи, вказівки щодо розроблення документації та проведення тестувань.
- Рекомендації щодо безпеки: методи та інструменти для забезпечення безпеки системи.
- Рекомендації щодо моніторингу та аналітики: відслідковування та моніторинг роботи системи, методи аналізу одержаних даних.
- Рекомендації щодо оновлення та розширення: процедури та вимоги для оновлення системи, рекомендації щодо розширення системи.
- Рекомендації щодо навчання користувачів: інструкції та матеріали для навчання користувачів системи.

Мета опису архітектурних рішень і рекомендацій щодо їх реалізації та підтримки в майбутньому полягає в забезпеченні ефективності, надійності та безпеки роботи системи впродовж її життєвого циклу. Важливо також забезпечити можливість

розширення та масштабування системи відповідно до змін потреб користувачів та ринкових умов.

Архітектурні паттерни для інтеграції ІС – це зразки дизайну, які можна використовувати для вирішення поширених проблем ІІС. Вони можуть допомогти в покращенні ефективності, надійності та безпеки ІІС.

Деякі з найпоширеніших архітектурних паттернів для інтеграції ІС такі:

- **Синхронна інтеграція:**
  - *Шина подій (Event Bus)* – використовується для забезпечення спільної комунікації між різними системами, де одна система може розповісти про певну подію, а інша система може підписатися на цю подію та отримувати повідомлення.
  - *Синхронний виклик (Synchronous Call)* – здійснюється безпосередньо з однієї системи до іншої, де система, що викликає, очікує на відповідь виконавчої системи.
- **Асинхронна інтеграція:**
  - *Асинхронне оброблення повідомлень (Message Queue)* – механізм передавання повідомлень між системами, де повідомлення зберігаються в черзі та обробляються після того, як виконується певна умова.
  - *Подіє-орієнтований підхід (Event-Driven Architecture)* – система заснована на сприйнятті та обробленні подій, які відбуваються в окремих системах.

Інтеграція з використанням сервісної орієнтованої архітектури (SOA).

Сервіс-орієнтована архітектура (Service-Oriented Architecture) – підхід до проєктування систем, що базується на розбитті функціональності на сервіси, які можуть бути використані різними системами для одержання необхідних даних або функцій.

Система керування сервісами (Service Registry) – реєстр сервісів, який дозволяє системам знайти та використовувати потрібні сервіси.

Для реалізації архітектурного паттерну SOA (Service-Oriented Architecture) необхідно забезпечити розроблення специфікацій сервісів та їх інтерфейсів, розроблення системи керування сервісами, використання стандартних протоколів для комунікації між сервісами та забезпечення моніторингу та аналізу роботи сервісів. Для паттерну EAI (Enterprise Application Integration) необхідно забезпечити використання платформи для інтеграції, розроблення маперів для трансформації даних між системами, використання стандартних протоколів для комунікації між системами та забезпечення моніторингу та аналізу роботи системи.

Нотації до опису архітектури

Нотація – це спосіб візуального подання архітектури системи та її компонентів у формі діаграм. Вони використовуються для комунікації між учасниками проєкту, документування архітектури системи та її подальшого аналізу.

Однією з найпопулярніших нотацій є UML, яка дозволяє описувати різні аспекти системи, враховуючи класи, об’єкти, взаємодію між компонентами та ін.

Інші популярні нотації містять BPMN для відображення бізнес-процесів, ArchiMate для побудови архітектури підприємств, DFD для моделювання потоків даних і C4 для подання архітектури розподілених інтегрованих систем.

Під час побудови документації на основі нотацій важливо зазначати пояснення до кожної діаграми та її компонентів, щоб забезпечити зрозумілість та узгодженість розуміння між учасниками проєкту.

Нотація C4: визначення та приклади

Нотація C4 – це проста та інтуїтивна нотація для опису архітектури програмного забезпечення, яка дозволяє описати систему на різних рівнях деталізації та забезпечити зрозумілість для різних аудиторій, враховуючи розробників, менеджерів і бізнес-аналітиків.

Основні принципи нотації C4

- Логічна структура. Нотація C4 фокусується на логічній структурі системи, а не на її фізичній реалізації. Це дозволяє описати систему незалежно від її конкретних технологій або платформ.
- Ієрархія. Нотація C4 пропонує ієрархію діаграм, які описують систему на різних рівнях деталізації. Це дозволяє фокусуватися на необхідних деталях для конкретної аудиторії.
- Відкритість. Нотація C4 є відкритою та гнучкою. Це дозволяє адаптувати її до конкретних потреб команди.

Нотація C4 складається **з чотирьох рівнів**:

- *Рівень контексту*. Діаграма контексту показує систему в контексті інших систем і зовнішніх акторів.
- *Рівень контейнерів*. Діаграма контейнерів показує внутрішню структуру системи, враховуючи контейнери та взаємодію між ними.
- *Рівень компонентів*. Діаграма компонентів показує детальну структуру компонентів системи та їх взаємодію.
- *Рівень коду*. Діаграма коду показує код, який реалізує компоненти системи.

Приклади застосування нотації C4

Розглянемо приклад застосування нотації C4 для опису архітектури веб-додатка.

1. Рівень контексту:

Діаграма контексту для веб-додатка може виглядати так:

Веб-додаток

- Користувачі.
- База даних.
- Внутрішні системи.

Ця діаграма показує, що веб-додаток взаємодіє з користувачами, базою даних і внутрішніми системами.

2. Рівень контейнерів:

Діаграма контейнерів для веб-додатка може виглядати так:

Веб-додаток

- Веб-сервер.
- База даних.
- Внутрішні системи.

Ця діаграма показує, що веб-додаток складається з веб-сервера, бази даних і внутрішніх систем.

3. Рівень компонентів:

Діаграма компонентів для веб-додатка може виглядати так:

Веб-додаток

- Фронтенд.
- Бекенд.

Ця діаграма показує, що веб-додаток складається з frontend і backend-компонентів.

4. Рівень коду:

Діаграма коду для веб-додатка може виглядати так:

Фронтенд

- HTML.
- CSS.
- JavaScript.

Бекенд

- PHP.
- MySQL.

Ця діаграма показує, що код веб-додатка написаний на PHP та MySQL.

Розглянемо **приклад** застосування нотації C4 для опису архітектури інтегрованої системи з мікросервісною архітектурою.

1. Рівень контексту

На цьому рівні можна відобразити системи та зовнішні сервіси, які взаємодіють з інтегрованою системою. Наприклад, в системі для онлайн-продажу товарів можна визначити інші системи компанії (компоненти складу, системи оплати), сервіси зовнішніх постачальників, веб-сайти клієнтів тощо.

2. *Рівень контейнеру*

На цьому рівні можна визначити окремі компоненти, які взаємодіють із зовнішніми системами. Наприклад, у системі для онлайн-продажу товарів можна виділити компоненти, які забезпечують взаємодію з базою даних, зовнішніми постачальниками, системами оплати тощо.

3. *Рівень компонентів*

На цьому рівні можна описати детальну структуру окремих компонентів, які взаємодіють з іншими компонентами. Наприклад, у системі для онлайн-продажу товарів можна описати компонент, який відповідає за оброблення замовлень, який містить декілька мікросервісів для одержання інформації про товари, їх відправку клієнту та оплату.

Існує три основні **рівні архітектури інтегрованих систем**:

- *Бізнес-рівень* описує бізнес-процеси та функціональні вимоги до системи. На цьому рівні зазвичай використовуються діаграми бізнес-процесів і моделі бізнес-об’єктів.
- *Рівень додатка* описує складові системи та їх взаємозв’язки. На цьому рівні зазвичай використовуються діаграми компонентів, діаграми класів і діаграми послідовності.
- *Рівень технічної інфраструктури* описує фізичну інфраструктуру системи та її конфігурацію. На цьому рівні зазвичай використовуються діаграми розгортання та діаграми мережі.

Три рівні архітектури взаємозв’язані між собою. Бізнес-рівень визначає функціональні вимоги до системи, які реалізуються на рівні додатка. Рівень додатка визначає компоненти системи, які розгортаються на рівні технічної інфраструктури.

Приклади

Одним із прикладів інтегрованої системи є система електронного документообігу. На рівні бізнес-архітектури ця система може мати такі бізнес-процеси, як створення,

оброблення та зберігання документів. На рівні архітектури додатків система може містити такі компоненти, як інформаційна система збереження документів, електронний підпис і модуль відправлення та одержання повідомлень. На рівні технологічної архітектури система може використовувати такі технології, як протоколи передачі даних, сервіси веб-сервера та бази даних.

Іншим прикладом інтегрованої системи є система управління ресурсами підприємства. На рівні бізнес-архітектури ця система може містити такі бізнес-процеси, як планування ресурсів, управління запасами та управління виробництвом. На рівні архітектури додатків система може містити такі модулі, як модуль управління запасами, модуль управління виробництвом та модуль управління кадрами. На рівні технологічної архітектури система може використовувати такі технології, як протоколи передачі даних, сервіси веб-сервера та бази даних.

Крім рівнів архітектури, важливим елементом архітектури інтегрованих систем є **середовище розгортання**, яке описує фізичну інфраструктуру, на якій працює система. Середовище розгортання може містити сервери, бази даних, мережеві комутатори та інші складові, які забезпечують роботу системи.

Для зображення середовища існування на архітектурній схемі можна використовувати діаграму розгортання (Deployment diagram) або діаграму контексту (Context diagram).

- **Діаграма розгортання** дозволяє показати фізичне розміщення компонентів системи (апаратного та програмного забезпечення) та залежності між ними. На діаграмі розгортання можна зобразити сервери, бази даних, мережеві з'єднання, програмне забезпечення та інші компоненти, що використовуються в середовищі.
- **Діаграма контексту**, зі свого боку, дозволяє показати зв'язки між системою та зовнішніми компонентами, такими як користувачі, інші системи, сервіси тощо. На діаграмі

контексту зображуються лише найважливіші зовнішні компоненти та їх зв'язки з системою.

На рисунку 3.1 подана діаграма розгортання програмного забезпечення системи обліку з поєднанням центрального серверу, бази даних і розумних контрактів.

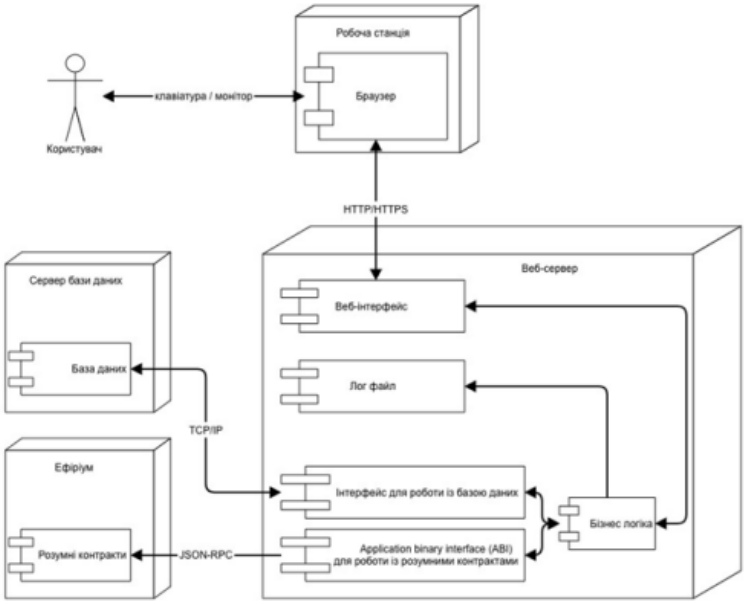

Рисунок 3.1 – Діаграма розгортання

У цьому прикладі центральний сервер розгорнутий на вузлі «Сервер» і має доступ до бази даних, яка розгорнута на вузлі «База даних». Розумні контракти запущені на блокчейні, що розгорнутий на вузлі «Блокчейн», та взаємодіють із центральним сервером за допомогою мережі Інтернет.

Інтегровані системи можуть бути розгорнуті в різних середовищах, залежно від потреб системи та специфіки діяльності організації. Основні **види середовищ**:

- *Локальне середовище* – це середовище, де всі компоненти системи працюють на одному комп'ютері або в локальній мережі. Це найпростіше середовище, яке не вимагає

значних витрат на апаратне та програмне забезпечення. Однак воно має обмеження в масштабованості та доступності.

- Хмарне середовище – це середовище, де компоненти системи розміщені на різних серверах, які розміщені в різних місцях і під'єднані до інтернету. Це середовище є більш масштабованим і доступним, ніж локальне середовище. Проте воно вимагає більших витрат на апаратне й програмне забезпечення.
- *Гібридне середовище* – це середовище, що поєднує локальні та хмарні компоненти системи. Це середовище є більш гнучким, ніж локальне або хмарне середовище. Воно дозволяє організації використовувати найкращі можливості кожного середовища.
- *Розподілене середовище* – це середовище, де компоненти системи розміщені на різних комп'ютерах і взаємодіють між собою через мережу. Це середовище є найбільш масштабованим і доступним, але також є найбільш складним в управлінні.

Приклади середовищ для функціонування ПС

- Обчислювальні хмари – це сервіс, який надає доступ до різноманітних ресурсів (обчислювальних потужностей, мережевих ресурсів, зберігання даних тощо) через мережу Інтернет. Приклади: Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform.
- Кластери – це група пов'язаних комп'ютерів, які працюють разом як єдине ціле. Кластери використовуються для розподілення навантаження та забезпечення високої доступності. Приклади: Apache Hadoop, Apache Spark.
- Контейнери – це технологія віртуалізації, яка дозволяє запускати додатки та сервіси в ізольованих середовищах. Контейнери є більш ефективними та легкими за використанням, ніж віртуальні машини. Приклади: Docker, Kubernetes.
- Мікросервісна архітектура – це архітектурний підхід, в якому програмне забезпечення розбивається на окремі

мікросервіси, які взаємодіють між собою через мережу. Кожен мікросервіс може бути розгорнутий в окремому середовищі. Приклади: Netflix, Uber.

Приклад. Мікросервісна архітектура Netflix

Netflix – це компанія, яка займається онлайн-стрімінгом відеоконтенту, і містить декілька інтегрованих інформаційних систем. Для цієї компанії була створена архітектура мікросервісів, що дозволяє їм швидко та ефективно розгортати нові функції та покращувати функціональність своєї платформи.

На високому рівні архітектура Netflix складається з трьох основних компонентів:

- Клієнтське програмне забезпечення – це пристрої, які використовують користувачі для перегляду відео, такі як телевізори, мобільні телефони та веб-браузери.
- API-сервери – це сервери, які обробляють запити від клієнтського програмного забезпечення.
- Бекенд-сервери – це сервери, які зберігають дані та обробляють запити від API-серверів.

Нижче рівнем, архітектура Netflix складається з мікросервісів, які мають відповідальність за конкретну функціональність. Кожен мікросервіс має свій власний інтерфейс програмування застосунків (API) і взаємодіє з іншими мікросервісами, використовуючи асинхронну комунікацію за допомогою механізмів, таких як Apache Kafka.

Netflix використовує понад 100 мікросервісів для обслуговування своєї платформи. Деякі з найважливіших мікросервісів мають:

- Рекомендаційний сервіс – це мікросервіс, який рекомендує користувачам відео, які вони можуть подивитися.
- Сервіс відтворення відео – це мікросервіс, який завантажує та відтворює відео для користувачів.
- Сервіс управління контентом – це мікросервіс, який використовується для управління каталогом відео Netflix.

Мікросервіси Netflix взаємодіють між собою за допомогою асинхронної комунікації. Це означає, що один

мікросервіс може надіслати запит іншому мікросервісу, не чекаючи відповіді. Це дозволяє мікросервісам працювати незалежно один від одного й масштабуватися незалежно.

Netflix працює на двох хмарах: Amazon Web Services (AWS) та Open Connect. AWS використовується для розміщення більшості мікросервісів і бекенд-серверів Netflix. Open Connect використовується для доставки відео до користувачів.

Інтерфейс користувача Netflix написаний на JavaScript. Netflix використовує ReactJS для створення інтерфейсу користувача, оскільки він швидкий, ефективний і модульний.

Інші діаграми

Діаграми послідовності, використання та класів є важливими інструментами для проєктування та описування функціональності системи. Вони допомагають розібратися в тому, як система повинна працювати, і виявляти можливі проблеми, що можуть виникнути під час реалізації системи. Ці діаграми допомагають архітекторам системи визначити, які компоненти системи будуть співпрацювати між собою та як будуть передаватися дані.

Діаграма послідовності – це графічне зображення взаємодії між об’єктами системи від початку до кінця певної операції. Вона показує послідовність повідомлень, що передаються між об’єктами, та час, коли вони відправляються та приймаються. На діаграмі послідовності можна показати, які сервіси між собою взаємодіють, які дані вони передають та які процеси відбуваються при цьому. Це допомагає забезпечити сумісність між системами та їх правильну інтеграцію.

Діаграма використання – це графічне зображення функціональної взаємодії між об’єктами системи. Вона показує, які об’єкти взаємодіють, що вони роблять та які повідомлення вони використовують.

Діаграма класів – це графічне зображення класів, їх атрибутів і методів, а також взаємозв’язків між ними. Вона показує структуру системи та залежності між її компонентами.

Під час проєктування інтегрованих інформаційних систем ці діаграми використовуються для визначення потреб користувачів, проєктування функціональності системи, розроблення бази даних та інтерфейсів користувача.

Наприклад, для інтегрованої системи електронної комерції можна використовувати діаграми послідовності для показу взаємодії між користувачем і системою під час оформлення замовлення, діаграму використання для визначення функціональності системи та діаграму класів для проєктування бази даних і взаємозв'язків між класами, такими як замовлення, користувач і товар.

На діаграмі послідовності для процесу онлайн-замовлення товарів на сайті інтернет-магазину можна показати взаємодію між користувачем, інтернет-магазином і платіжною системою під час процесу онлайн-замовлення товарів. Користувач вибирає товари на сайті, додає їх до кошика, вводить адресу доставки та обирає спосіб оплати. Інтернет-магазин оброблює замовлення, перевіряє наявність товарів на складі та здійснює оплату через платіжну систему. Після успішної оплати замовлення відправляється на адресу доставки.

На діаграмі використання для системи електронного банкінгу необхідно показати можливі дії користувачів у системі електронного банкінгу. Користувачі можуть перевіряти баланс свого рахунку, здійснювати перекази коштів, оплачувати рахунки, налаштовувати повідомлення про транзакції тощо. Система також може автоматично відправляти повідомлення про стан рахунку та транзакції на електронну пошту або мобільний телефон користувача.

Тема 4. Інтеграція на основі XML, JSON

Лекція 6. Інтеграція на основі XML

Під час розроблення веб-сервісів важливо враховувати формат даних, якими обмінюються клієнтські застосунки та сервіс. Цей формат повинен бути простим, читабельним і зв'язним.

Об'єкти моделі даних зазвичай пов'язані між собою. Ці відносини повинні відображатися в способі подання даних клієнтським застосункам. Наприклад, у сервісі обговорень пов'язані ресурси можуть містити тему обговорення, її атрибути та посилання на відповіді.

Щоб надати клієнтським застосункам можливість запитувати конкретний тип вмісту, проєктують сервіс так, щоб він використовував вбудований HTTP-заголовок Accept. Значення цього заголовка є MIME-типом, наприклад, application/json, application/xml або application/xhtml+xml.

Означення об'єктів даних і обмін даними

Обмін даними в комп'ютерних системах зазвичай відбувається з використанням структурованих даних. Ці дані зберігаються, обробляються й передаються між пристроями та системами. Проте комп'ютерні системи можуть мати різні апаратні архітектури та різні способи подання даних.

Один із способів організації формату передачі даних – це подання їх у вигляді тексту. Це дозволяє забезпечити сумісність між різними системами, але збільшує обсяг даних, які передаються, і потребує додаткових витрат на їх оброблення.

Додатковою перевагою текстового подання є можливість передавати (і зберігати) не лише самі дані, а і їх опис (метадані). Тому існуючі та використовувані текстові подання (серіалізація) даних називається не форматом даних, а мовою опису даних.

Перевірка даних на відповідність схемі

Під час обміну текстовим поданням даних відправник та отримувач повинні однаково сприймати ці дані. Для цього дані перед обробленням бажано перевірити на відповідність певній

схемі (валідність). Така перевірка можлива лише за наявності додаткової інформації, яка б описувала схему.

Наявність механізмів такої перевірки може стати важливим аргументом під час вибору того чи іншого способу серіалізації.

На рисунку 4.1 наведено приклад XML-документа, який описує тему обговорення в форумі.

```
<?xml version="1.0"?>
<discussion date="2023-08-31" topic="Новий випуск
WordPress">
  <comment>Цей випуск включає в себе багато нових функцій і
поліпшень.</comment>
  <replies>
    <reply from="joe@mail.com"
href="/discussion/topics/Новий випуск WordPress/joe">
      Я спробував новий редактор публікацій, і він мені
дуже сподобався.
    </reply>
    <reply from="bob@mail.com"
href="/discussion/topics/Новий випуск WordPress/bob">
      Мені теж сподобалися нові можливості персоналізації.
    </reply>
  </replies>
</discussion>
```

Рисунок 4.1 – Приклад XML-документа

Передавання XML у веб-сервісах

Під час розроблення веб-сервісів важливо враховувати формат даних, якими обмінюються клієнтські застосунки та сервіс. Цей формат повинен бути простим, читабельним і зв’язним.

Об’єкти моделі даних зазвичай пов’язані між собою. Ці відносини повинні відображатися в способі подання даних клієнтським застосункам. Наприклад, у сервісі обговорень пов’язані ресурси можуть включати в себе тему обговорення, її атрибути та посилання на відповіді.

Щоб надати клієнтським застосункам можливість запитувати конкретний тип вмісту, проєктують сервіс так, щоб він використовував вбудований HTTP-заголовок Accept.

Значення цього заголовка є MIME-типом, наприклад, application/json, application/xml або application/xhtml+xml.

Обмін даними в комп'ютерних системах зазвичай відбувається з використанням структурованих даних. Ці дані зберігаються, обробляються і передаються між пристроями та системами. Проте комп'ютерні системи можуть мати різні апаратні архітектури та різні способи подання даних.

Один із способів організації формату передачі даних – це подання їх у вигляді тексту. Це дозволяє забезпечити сумісність між різними системами, але збільшує обсяг даних, які передаються, і потребує додаткових витрат на їх оброблення.

Додатковою перевагою текстового подання є можливість передавати (і зберігати) не лише самі дані, а і їх опис (метадані). Тому існуючі та використовувані текстові подання (серіалізація) даних називається не форматом даних, а мовою опису даних.

Перевірка даних на відповідність схемі

Під час обміну текстовим поданням даних відправник та отримувач повинні однаково сприймати ці дані. Для цього дані перед обробленням бажано перевірити на відповідність певній схемі (валідність). Така перевірка можлива лише за наявності додаткової інформації, яка б описувала схему.

Наявність механізмів такої перевірки може стати важливим аргументом під час вибору того чи іншого способу серіалізації.

На рисунку 4.2 наведено приклад XML-документа, який описує тему обговорення в форумі.

```
<?xml version="1.0"?>
<discussion date="2023-08-31" topic="Новий випуск WordPress">
  <comment>Цей випуск включає в себе багато нових функцій і поліпшень.</comment>
  <replies>
    <reply from="joe@mail.com" href="/discussion/topics/Новий випуск WordPress/joe"
      Я спробував новий редактор публікацій, і він мені дуже сподобався.
    </reply>
    <reply from="bob@mail.com" href="/discussion/topics/Новий випуск WordPress/bob"
      Мені теж сподобалися нові можливості персоналізації.
    </reply>
  </replies>
</discussion>
```

Рисунок 4.2 – Приклад XML-документа

Цей документ містить такі елементи: <discussion> – корінь документа, який описує тему обговорення; <date> – атрибут, який містить дату створення теми; <topic> – атрибут, який містить назву теми; <comment> – елемент, який містить коментар до теми; <replies> – елемент, який містить список відповідей на коментар; <reply> – елемент, який описує одну відповідь; <from> – атрибут, який містить адресу електронної пошти автора відповіді; <href> – атрибут, який містить посилання на сторінку з відповіддю.

Загальне подання XML

XML – eXtensible Markup Language – метамова, що надає гнучкий інструментарій формування документів, призначених для опису структур даних для їх машинного оброблення. XML є еволюцією іншої мови SGML (Standard Generalised Markup Language) від якої також з'явилася мова HTML. XML використовується на практиці для опису складних структур даних (опис переліків полів даних, типів даних, способів їх оброблення), обміну цими описами між комп'ютерними системами та збереження цих даних.

XML-документи являють собою текстові файли, які складаються з елементів, що позначають дані. Кожен елемент починається з відкриваючого тегу, закінчується закриваючим тегом і може містити в собі інші елементи або текстові значення.

XML-теги можуть мати атрибути, які надають додаткову інформацію про елемент. Атрибути позначаються у вигляді пар ключ-значення, розділених знаком рівності.

XML-документи можуть бути структуровані у вигляді дерева, де елементи можуть вкладатися один в одного. Це дозволяє подавати складні структури даних у зручному для оброблення вигляді.

Структура документа:

- **Теги** (Tag) – будь-яке слово латиницею, наприклад <Tag>, що показує на розмітку даних, тобто загорнутий у < та >.

- **Зміст тегів (контент)** – все, що розміщене між відкритими <Tag> та закритими </Tag> тегами.
- **Атрибути тегів** – частини, які описують властивості (атрибути) об'єкта, у форматі <Tag attribute="attribute value"/>.
- Розділ даних DTD (CDATA) – блок даних, що не містить елементів розмітки (лише корисне навантаження), наприклад, <![CDATA[*будь-які дані*]]>.
- Розділ коментарів, наприклад <!-- Commentaries -->.
- Метадані в заголовку - дані про документ, наприклад <?xml version="1.0" encoding="UTF-8"?>

XML елемент – одиниця опису XML, який обгорнутий тегом зі змістом, приклади:

- <price>1.29</price> – елемент price із значенням;
- <emptyTag></emptyTag> – пустий елемент, або те саме <emptyTag/>;
- <price type="wholesale">1.29</price> – атрибут type із значенням wholesale.

Назви тегів XML:

- можуть містити літери, цифри, дефіси, підкреслення та крапки, але не можуть містити пробіли;
- не можуть починатися з літер xml;
- повинні починатися з літери або підкреслення;
- чутливі до регістру, тобто <Letter> відрізняється від <letter>.
- можуть містити нелатинські літери.

Для розділення частин назви тегу рекомендується використовувати символ _ або різний регістр літер.

Теги є мітками для виділення певного іменованого змісту (content), який вони обрамляють. Зміст може бути звичайним текстом або містити інші XML-елементи. Наприклад, у наведеному вище прикладі значення поіменованої сутності price дорівнює 30.00.

Текстовий контент може містити будь-які символи, за винятком службових.

На відміну від HTML, у XML пробіли не видаляються, а мультирядкові тексти розділяються символом LF.

Службові символи в XML можуть спричинити помилку, якщо їх розмістити всередині елемента. Щоб уникнути цієї помилки, спеціальні символи можна замінити посиланням на сутність. У XML є 5 попередньо означених посилань на сутності (табл. 4.1).

Таблиця 4.1 – Посилання на сутності

Необхідний символ	**Замінник**
<	&lt;
>	&gt;
&	&amp;
'	&apos;
"	&quot;

Атрибути – це властивості поіменованих сутностей, які записують у форматі ключ="значення" або ключ='значення'. У тегу може бути довільна кількість атрибутів, які записуються через пробіл. У наведеному вище прикладі в одного з тегів book є два атрибути: category="computer" і subcategory="web".

XML-елементи – це теги з їх змістом. Вкладеність елементів в інші елементи формує перегорнуте догори ногами дерево, яке починається з одного кореневого (root) елементу, який має дочірні (child) елементи, які так само можуть мати дочірні елементи і так далі.

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

Порожній елемент – це елемент, який не має вмісту. У XML його можна зазначити так:

<element></element>

або так:

<element />

Код XML з bookstore можна відобразити графічно як на рисунку 4.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is a comment -->
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```

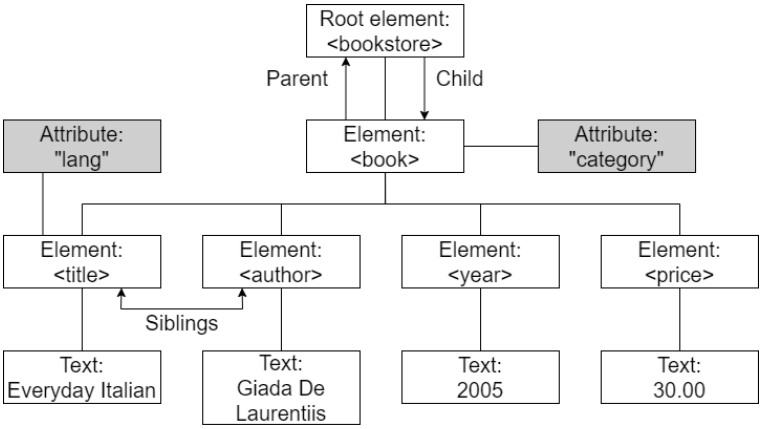

Рисунок 4.3 – Графічне відображення XML-документа

Окрім даних, XML-документ може містити метадані в заголовку (пролозі), який визначає версію XML і кодування символів. Заголовок XML виглядає так:

<?xml version="1.0" encoding="UTF-8"?>.

Заголовок XML повинен розміщуватися в першому рядку документа і всі символи в ньому повинні бути в нижньому регістрі. Окрім атрибутів version і encoding, заголовок може містити атрибут standalone="yes", який свідчить про те, що в документі не використовується DTD (описано нижче).

Коментарі в XML починаються з символів <!-- і закінчуються символами -->. Два тире всередині коментаря не допускаються.

Для того щоб передавати текст без заміни заборонених символів посиланнями на сутності, можна використовувати розділи CDATA.

- Розділи CDATA можуть з’являтися всередині вмісту елемента і дозволяють відображати, наприклад, символьні літерали < та &.
- Розділ CDATA починається з послідовності символів <![CDATA[і закінчується послідовністю символів]]>.

- Між двома послідовностями символів XML-процесор ігнорує всі символи розмітки, такі як <, > і &. Єдина розмітка, яку XML-процесор розпізнає всередині розділу CDATA, – це послідовність закриваючих символів]]>.
- Послідовність символів, яка закінчує розділ CDATA]]>, не повинна з'являтися всередині вмісту елемента. Натомість завершальний символ > має бути екранований за допомогою відповідної сутності >.
- Розділи CDATA не можуть бути вкладеними.

Приклад розбору в Google Apps Scripts

Для роботи XML в усіх сучасних мовах і середовищах програмування надаються певні бібліотеки або сервіси. У Google Apps Scripts наприклад, для роботи з XML є окремий вбудований сервіс XML Service. Цей сервіс дозволяє сценаріям аналізувати, переглядати та програмно створювати документи XML.

Приклад для демонстрації роботи деяких методів сервісу та його класів поданий на рисунку 4.4.

Приклад виводу цього коду буде таким:

John Doe
30
years
Main Street
CA

У цьому прикладі ми виконуємо такі кроки:

1. Отримуємо текст XML-документа.
2. Парсуємо XML-документ за допомогою методу parse.
3. Друкуємо значення кореневого елемента.
4. Друкуємо значення елемента name.
5. Друкуємо атрибут unit елемента age.
6. Друкуємо значення елемента street.

7. Друкуємо значення атрибута code елемента state.

Для отримання об’єкта з XML використовується метод parse в який як аргумент вставляється текстовий зміст, а результатом є об’єкт Document. Далі документ можна аналізувати або змінювати, використовуючи його методи. Можна дістатися до всіх вузлів через метод getAllContent() або до елементів (об'єкти класу Element), починаючи з кореневого методу getRootElement(), а потім поступово вглиб до всіх дочірніх методу getChildren() або вказуючи назву елементів, до яких потрібно дістатися. Методи класу Element надають можливість дістатися до значення, змісту та атрибутів елементу.

```
function parseXml() {
  // Отримуємо текст XML-документа
  var xmlText = `
    <person>
      <name>John Doe</name>
      <age>30</age>
      <address>
        <street>Main Street</street>
        <city>San Francisco</city>
        <state>CA</state>
      </address>
    </person>
  `;

  // Парсуємо XML-документ
  var document = Utilities.xml.parse(xmlText);

  // Друкуємо значення кореневого елемента
  Logger.log(document.getRootElement().getValue());

  // Друкуємо значення елемента name
  var nameElement = document.getRootElement().getChild("name");
  Logger.log(nameElement.getValue());

  // Друкуємо атрибут age елемента
  var ageElement = document.getRootElement().getChild("age");
  Logger.log(ageElement.getAttribute("unit"));

  // Друкуємо значення елемента street
  var addressElement = document.getRootElement().getChild("address");
  var streetElement = addressElement.getChild("street");
  Logger.log(streetElement.getValue());

  // Друкуємо значення атрибута state елемента address
  var stateElement = addressElement.getChild("state");
  Logger.log(stateElement.getAttribute("code"));
}
```

Рисунок 4.4 – Демонстрація роботи деяких методів сервісу та його класів

Простір імен XML – це колекція імен елементів і атрибутів XML, унікальних у межах цього простору. Ідентифікатор простору імен – це унікальний URI, який використовується для посилання на цей простір імен.

Простори імен оголошуються як атрибути будь-якого елемента документа XML за таким синтаксисом:

<someElement xmlns:prefix=namespace/>,

де:

- prefix – префікс, який використовується для скорочення посилання на цей простір імен далі в документі XML.
- namespace – ідентифікатор простору імен.

Якщо префікс не зазначено, то елемент буде використовувати простір імен за замовчуванням.

Щоб використовувати елемент або атрибут із певного простору імен, необхідно зазначити префікс цього простору імен перед його ім’ям.

Наприклад, у наступному документі XML-елемент student розміщений у просторі імен https://www.develop.com/student, а атрибут id розміщений у просторі імен за замовчуванням (рис. 4.5).

```
<d:student xmlns:d='https://www.develop.com/student' id='3235329'>
  <name>Jeff Smith</name>
  <language>C#</language>
  <rating>35</rating>
</d:student>
```

Рисунок 4.5 – Демонстрація оголошення простору імен

Щоб одержати доступ до елемента student із простору імен https://www.develop.com/student, можна використовувати код із рисунка 4.6.

```
let document = XmlService.parse(xmlcontent);
let root = document.getRootElement();
let student = root.getChild('d:student');
```

Рисунок 4.6 – Доступ до елемента student

Атрибути не пов’язуються автоматично з ідентифікатором простору імен за замовчуванням. Щоб пов’язати атрибут із простором імен, необхідно зазначити префікс.

Розбір і оброблення XML-документів

XML-документи можуть бути перетворені в об'єкти, з якими зручно працювати в коді програми, за допомогою парсерів. **Парсер** – це програма, яка використовує специфічну мову програмування для розбору XML-документів.

У середовищі програмування Google Apps Script для розбору XML-документів використовується метод parse() об'єкта XmlService. Цей метод бере як аргумент вміст XML-документа і повертає об'єкт Document. Об'єкт Document являє собою дерево вузлів, що відповідає структурі XML-документа.

Є два основних підходи до розбору XML-документів:

1. DOM (Document Object Model) – парсер, який повністю завантажує в оперативну пам'ять структуру XML-документа. Цей підхід дозволяє працювати з об'єктом «цілком», але вимагає великої кількості пам'яті й процесорного часу.
2. SAX (Simple API for XML) – парсер, який обробляє XML-документ методом траверсингу (руху від елементу до елементу). Цей підхід не вимагає великої кількості пам'яті й працює набагато швидше, ніж DOM, але не дозволяє працювати з об'єктом «цілком».

У методі parse() об'єкта XmlService використовується підхід DOM. На виході методу формується об'єкт Document, який має набір методів для доступу до розпарсеного об'єкта.

XSL, трансформація XSLT

XSL (Extensible Stylesheet Language) – це сімейство рекомендацій для означення перетворень і подання XML-документів. XSL складається з чотирьох частин:

- XPath – мова для виразів, яка використовується XSLT для доступу до частин XML-документа;
- XSLT – мова для опису перетворень XML-документів;
- XSL-FO -XML – словник для описання семантики форматування;
- XQuery – мова для запитів документів XML;
- XPath.

XPath використовує вирази шляху для вибору вузлів або наборів вузлів у документі XML. Ці вирази шляху дуже схожі на вирази, які ви бачите під час роботи з традиційною файловою системою комп'ютера.

У XPath існує сім типів вузлів: елемент (element); атрибут (attribute); текст (text); простір імен (namespace); інструкція оброблення (processing-instruction); коментар (comment); кореневий вузол (root node).

Документи XML розглядаються як дерева вузлів. Верхній елемент дерева називається кореневим елементом. Вузол вибирається за шляхом або кроками.

У таблиці 4.2 подані найбільш корисні вирази шляху.

Таблиця 4.2 – Вирази шляху

Вираз	**Опис**
nodename	Вибирає всі вузли з назвою nodename
/	Вибирає з кореневого вузла
//	Вибирає вузли в документі з поточного вузла, які відповідають виділенню незалежно від того, де вони розміщені
.	Вибирає поточний вузол (контекстний)
..	Вибирає батьківський елемент для поточного вузла
@	Вибирає атрибути
comment( )	Вибирає коментарі
text( )	Вибирає текст CDATA
processing-instruction( )	Вибирає інструкція оброблення

У таблиці 4.3 подані приклади.

Таблиця 4.3 – Приклади

Вираз	**Результат**
bookstore	Вибирає всі вузли з назвою bookstore
/bookstore	Вибирає кореневий елемент bookstore. Зверніть увагу, якщо шлях починається зі слеша /, він завжди мвє абсолютний шлях до елемента!
bookstore/book	Вибирає всі елементи book, які є дочірніми для bookstore
//book	Вибирає всі елементи book незалежно від того, де вони розміщені в документі
bookstore//book	Вибирає всі елементи book, які є нащадками елемента bookstore, незалежно від того, де вони розміщені під елементом bookstore
//@lang	Вибирає всі атрибути з назвою lang

Використання XML у комбінації для забезпечення взаємодії з веб-сервісами

XML можна використовувати для **передавання повідомлень** між різними програмами. Це відбувається шляхом створення документа XML, який містить ці повідомлення. Документ XML потім передається з однієї програми до іншої.

Прикладом використання XML для передавання повідомлень є обмін повідомленнями між клієнтом і сервером. Клієнт може створити документ XML, який містить дані повідомлення, наприклад, ім’я користувача та пароль. Документ XML потім передається серверу. Сервер може розпарсувати документ XML і обробити повідомлення.

На рисунку 4.7 поданий приклад того, як можна використовувати XML для передачі повідомлень між веб-застосунками.

```
// Створення клієнта для веб-застосунку
let client = new XMLHttpRequest();

// Створення документа XML
let document = new DOMDocument();

// Додавання даних повідомлення до документа XML
document.appendChild(document.createElement("message"));
document.documentElement.appendChild(document.createTextNode("Hello, world!"));

// Налаштування методу HTTP
client.open("POST", "https://example.com/messages");

// Додавання документа XML до запиту
client.setRequestHeader("Content-Type", "application/xml");
client.send(document.toXMLString());
```

Рисунок 4.7 – Використання XML для передавання повідомлень між веб-застосунками

У цьому прикладі клієнт створює документ XML, який містить повідомлення "Hello, world!". Документ XML потім передається веб-застосуванню за допомогою HTTP-методу POST. Веб-застосунок розпарсовує документ XML і обробляє повідомлення.

XML також можна використовувати для **викликів до веб-сервісів**. Це відбувається шляхом створення документа XML, який містить дані запиту. Документ XML потім передається веб-сервісу. Веб-сервіс може розпарсувати документ XML і обробити запит.

Прикладом використання XML для викликів до веб-сервісів є одержання інформації про продукт від веб-сервісу. Клієнт може створити документ XML, який містить запит на інформацію про продукт. Документ XML потім передається веб-сервісу. Веб-сервіс повертає документ XML, який містить інформацію про продукт.

XML і HTTP можуть використовуватися разом для забезпечення взаємодії з веб-сервісами. Веб-сервіс – це

програмний інтерфейс, який доступний через Інтернет. Веб-сервіси часто використовуються для надання доступу до даних або обчислень.

Для взаємодії з веб-сервісом, який використовує XML, клієнт повинен створити запит у форматі XML. Запит може містити дані, які клієнт хоче надіслати веб-сервісу, або команди, які клієнт хоче виконати.

Запит клієнта пересилається веб-сервісу за допомогою HTTP-методу POST. HTTP-метод POST використовується для відправлення даних на сервер.

Веб-сервіс одержує запит клієнта і обробляє його. Відповідь веб-сервісу також пересилається клієнту за допомогою HTTP-методу POST. HTTP-метод POST використовується для одержання даних від сервера.

Відповідь веб-сервісу містить дані, які веб-сервіс хоче передати клієнту. Дані можуть бути подані в форматі XML, JSON або будь-якому іншому форматі, який підтримує веб-сервіс.

На рисунку 4.8 поданий приклад, що показує, як можна використовувати XML і HTTP-методи для взаємодії з веб-сервісом, який надає доступ до інформації про продукти.

```
// Створення клієнта для веб-сервісу
let client = new XMLHttpRequest();
// Встановлення методу HTTP
client.open("GET", "https://example.com/products");
// Відправка запиту
client.send();
// Обробка відповіді
client.onload = function() {
  // Отримання об'єкта XML з відповіді
  let responseXML = client.responseXML;
  // Перебір елементів "product"
  for (let productElement of responseXML.getElementsByTagName("product")) {
    // Використання даних про продукт
    // ...
  }
};
```

Рисунок 4.8 – Використання XML і HTTP-методів для взаємодії з веб-сервісом

У цьому прикладі клієнт використовує HTTP-метод GET для одержання інформації про продукти від веб-сервісу. Веб-сервіс повертає документ XML, який містить інформацію про продукти. Клієнт розпарсовує документ XML і використовує одержані дані.

Лекція 7. Інтеграція на основі JSON

JSON (JavaScript Object Notation) – це текстовий формат обміну даними, який використовується для передачі та зберігання структурованих даних між різними системами.

JSON є незалежним від платформи та мови програмування, тому є досить популярним для використання у веб-програмуванні та програмному забезпеченні загалом.

Об'єкти в JSON – це набори пар ключ-значення, де ключ – це рядок, а значення – це будь-який тип даних JSON.

Масиви в JSON – це впорядковані списки значень, де значення можуть бути різних типів даних JSON.

JSON підтримує такі типи даних:

- рядки (strings);
- числа (numbers);
- булеві значення (booleans);
- масиви (arrays);
- об'єкти (objects);
- null.

Дані, подані у форматі JSON, можна інтегрувати в різні системи й сервіси за допомогою таких методів:

1. **API (Application Programming Interface)** – це інтерфейс програмування, який дозволяє системам і сервісам взаємодіяти між собою та обмінюватися даними. API може бути побудований на основі формату JSON та дозволяє взаємодіяти з різними системами та сервісами через мережу Інтернет.

2. **Бібліотеки та фреймворки** – це засоби програмування, які дозволяють розробникам легко інтегрувати дані, подані у форматі JSON, в їх програми та сервіси. Наприклад, для мов програмування, таких як Python, JavaScript, Java, є багато бібліотек і фреймворків, які дозволяють розробникам працювати з даними в форматі JSON.

3. **Послуги оброблення даних** – деякі компанії надають послуги оброблення даних у форматі JSON, наприклад, зберігання даних у хмарі, пошук та аналіз даних. Ці послуги дозволяють розробникам і бізнес-користувачам зручно

працювати з даними в форматі JSON без необхідності у власних інфраструктурах.

4. **Мікросервісна архітектура** – це архітектурний підхід, який дозволяє розбити більші системи на менші, самодостатні мікросервіси, які можуть взаємодіяти між собою та обмінюватися даними в форматі JSON. Цей підхід дозволяє розробникам підтримувати систему більш гнучкою та легко масштабувати її.

Приклад використання Facebook Graph API для відображення профілю користувача.

Припустимо, ви хочете створити веб-додаток, який відображає профіль користувача Facebook. Для цього ви можете використовувати Facebook Graph API.

Facebook Graph API дозволяє вам одержувати дані у форматі JSON, які містять інформацію про користувача, а також його публікації, друзів, групи та інші дані.

Приклад коду, який можна використовувати для одержання даних із Facebook Graph API поданий на рисунку 4.9.

```
Python
import requests
# Отримуємо ключ API
API_KEY = "YOUR_API_KEY"
# Отримуємо дані з Facebook Graph API
url = "https://graph.facebook.com/YOUR_USER_ID?fields=name,email,picture&access_token={}".format(API_K
response = requests.get(url)
# Перетворюємо дані в формат JSON
data = response.json()
# Виводимо дані на екран
print(data)
```

Рисунок 4.9 – Одержання даних із Facebook Graph API

Цей код спочатку одержує ключ API з Facebook. Потім він використовує ключ API для одержання даних із Facebook Graph API. Приклад даних, які повертає API поданий на рисунку 4.10.

```
JSON
{
  "id": "YOUR_USER_ID",
  "name": "John Doe",
  "email": "johndoe@example.com",
  "picture": "https://graph.facebook.com/YOUR_USER_ID/picture?type=large
}
```

Рисунок 4.10 – Приклад даних, які повертає API

Ці дані містять основну інформацію про користувача, а саме:

- id: Ідентифікатор користувача.
- name: Ім’я користувача.
- email: Адреса електронної пошти користувача.
- picture: URL-адреса зображення профілю користувача.

Ви можете використовувати ці дані для відображення профілю користувача.

Приклад коду, який ви можете використовувати для відображення профілю користувача John Doe поданий на рисунку 4.11.

```
Python
import requests

# Отримуємо ключ API
API_KEY = "YOUR_API_KEY"

# Отримуємо дані з Facebook Graph API
url = "https://graph.facebook.com/10000123456789?fields=name,email,picture&access_token={}".format(API_K
response = requests.get(url)

# Перетворюємо дані в формат JSON
data = response.json()

# Відтворюємо сторінку профілю
html = """
<html>
<head>
  <title>Профіль користувача</title>
</head>
<body>
  <h1>Ім'я: {}</h1>
  <h2>Електронна пошта: {}</h2>
  <img src="{}" alt="Зображення профілю">
</body>
</html>
""".format(data['name'], data['email'], data['picture'])

# Виводимо сторінку профілю
print(html)
```

Рисунок 4.11 – Відображення профілю користувача John Doe

Цей код одержує дані з Facebook Graph API, а потім відтворює сторінку профілю, на якій відображаються ім'я, адреса електронної пошти та зображення профілю користувача John Doe.

Методи оброблення даних у форматі JSON

Парсинг – це процес перетворення рядка, який містить дані у форматі JSON, в структуру даних, яку можна легко обробляти у програмі (рис. 4.12).

```
Python
import json
json_string = '{"name": "John Smith", "age": 30, "city": "New York"
data = json.loads(json_string)
print(data["name"]) # виведе "John Smith"
```

Рисунок 4.12 – Приклад парсингу даних

Серіалізація – це процес перетворення структури даних у формат JSON у рядок, який можна відправити або зберегти в файл (рис. 4.13).

```
Python
import json

data = {
    "name": "John Smith",
    "age": 30,
    "city": "New York"
}

with open('data.json', 'w') as outfile:
    json.dump(data, outfile)
```

Рисунок 4.13 – Приклад серіалізації

Валідація – це процес перевірки того, чи відповідає рядок, який містить дані у форматі JSON, синтаксису цього формату та чи містить він правильні дані (рис. 4.14).

```
JavaScript
let json_string = '{"name": "John Smith", "age": 30, "city": "New York"}';

try {
  JSON.parse(json_string);
  console.log('Valid JSON');
} catch (e) {
  console.log('Invalid JSON');
}
```

Рисунок 4.14 – Приклад валідації

Маніпулювання даними – це процес оброблення та зміни даних у форматі JSON, таких як додавання, видалення або зміна елементів (рис. 4.15).

```
Python
data = {
    "name": "John Smith",
    "age": 30,
    "city": "New York"
}
# Додавання нового поля
data["email"] = "john.smith@example.com"
# Вилучення поля
del data["city"]
# Оновлення поля
data["age"] = 31
```

Рисунок 4.15 – Приклад маніпулювання даними

Варіанти створення файлу JSON, що містить дані для інтеграції:

- **Створення файлу вручну:** Ви можете створити файл JSON вручну, використовуючи будь-який текстовий редактор, такий як Notepad, Sublime Text, або Visual Studio Code. Файл повинен містити дані у форматі JSON, розділені комами та взяті в фігурні дужки. Наприклад, файл може виглядати так:

```
{
  "name": "John Smith",
  "age": 30,
  "city": "New York"
}
```

- **Використання програми або сервісу для генерації файлу:** існують спеціальні програми та онлайн-сервіси, які дозволяють генерувати файли JSON з даними. Наприклад, сервіс JSON Generator (https://www.json-generator.com/), який дозволяє створювати зразки даних у форматі JSON на основі шаблонів.
- **Збереження даних із бази даних у форматі JSON:** якщо використовується база даних, така як MySQL або PostgreSQL, можна зберегти дані у форматі JSON, використовуючи відповідний запит. Наприклад, можна використати команду SELECT для вибору даних із бази даних і використати функцію JSON_OBJECT для перетворення їх у формат JSON.

У таблиці 4.4 подані приклади відповідності завдань інтеграції до методів.

Таблиця 4.4 – Приклади відповідності завдань інтеграції до методів

Приклад	**Метод оброблення**	**Опис**
1	**2**	**3**
Інтеграція з веб-сервісами	Парсинг, серіалізація	Веб-сервіси дозволяють інтегрувати дані у форматі JSON у різні системи та сервіси. Наприклад, при розроблення додатка на основі JavaScript, можна використовувати бібліотеки Axios або jQuery, для звернення до REST API та одержання даних у форматі JSON
Інтеграція з базами даних	Парсинг, серіалізація	Бази даних, такі як MySQL або PostgreSQL, можуть зберігати дані у форматі JSON та дозволяють взаємодіяти з даними за допомогою SQL-запитів Наприклад, можна створити додаток на основі Node.js, який одержує дані з бази даних і повертає їх у форматі JSON

Продовження таблиці 4.4.

1	2	3
Інтеграція з хмарами даних	Парсинг, серіалізація	Хмарні сервіси, такі як Amazon S3 або Google Cloud Storage, дозволяють зберігати дані у форматі JSON та взаємодіяти з ними за допомогою API. Наприклад, можна використати сервіс Amazon S3 для збереження та обміну даними у форматі JSON між різними додатками та сервісами
Інтеграція з додатками для оброблення даних	Парсинг	Додатки для оброблення даних, такі як Microsoft Excel або Google Sheets, можуть імпортувати та експортувати дані у форматі JSON. Наприклад, можна імпортувати дані у форматі JSON в Google Sheets та використовувати їх для створення звітів та аналітики
Інтеграція з соціальними мережами	Парсинг, серіалізація	Багато сайтів і додатків інтегруються з соціальними мережами, такими як Facebook, Twitter або Instagram, для одержання доступу до профілю користувача або для публікації повідомлень на стіні. У цьому разі дані профілю користувача та інші деталі передаються у форматі JSON між сайтом або додатком і соціальною мережею
Інтеграція з платіжними системами	Парсинг, серіалізація	Платіжні системи, такі як PayPal, Stripe або Square, також підтримують інтеграцію даних у форматі JSON. Це дозволяє сайтам і додаткам здійснювати платежі через ці системи та одержувати повідомлення про платежі у форматі JSON

Інтеграція з мобільними додатками	Парсинг, серіалізація	Мобільні додатки, такі як додатки для здоров'я та фітнесу або додатки для онлайн-покупок, можуть використовувати інтеграцію даних у форматі JSON для одержання даних із серверів і передачі даних між додатком і сервером

Інструменти та бібліотеки у таблиці 4.5 дозволяють перетворювати дані у форматі JSON у різних мовах програмування, а також здійснювати роботу з цими даними, враховуючи створення, читання, оновлення та видалення даних у форматі JSON.

Таблиця 4.5 – Інструменти та бібліотеки для перетворення даних у форматі JSON

Мова/Інструмент	**Назва**	**Опис**
JavaScript	JSON.parse()	Вбудована функція в JavaScript для розбору JSON-рядків
JavaScript	JSON.stringify()	Вбудована функція в JavaScript для перетворення об'єктів JavaScript на JSON-рядки
Python	json	Бібліотека Python для роботи з JSON
Java	Jackson	Бібліотека Java для роботи з JSON
C#	Newtonsoft.Json	Бібліотека C# для роботи з JSON
PHP	json_encode() та json_decode()	Вбудовані функції PHP для перетворення між JSON-рядками та об'єктами PHP
Ruby	JSON	Вбудована бібліотека Ruby для роботи з JSON
Go	encoding/json	Бібліотека Go для роботи з JSON

Swift	Codable	Робота з JSON в Swift за допомогою протоколу Codable
.NET	System.Text.Json	Бібліотека .NET для роботи з JSON, вбудована з .NET Core 3.0 та вище

Розглянемо *приклад інтеграції даних у форматі JSON із веб-сайтом*. Припустимо, що існує веб-сайт для онлайн-магазину, і необхідно вивести на сторінці замовлення список товарів, які є в кошику користувача. Для того щоб інтегрувати дані у форматі JSON із веб-сайтом, необхідно спочатку одержати ці дані. У цьому разі можна зробити це за допомогою веб-сервісу, який надає список товарів, що є в кошику користувача. Цей сервіс може бути реалізований, наприклад, на основі REST API. Отримавши дані у форматі JSON, можна парсувати їх, щоб одержати доступ до інформації про кожний товар. Після цього можна використовувати цю інформацію для відображення списку товарів на сторінці замовлення. Можливий код цієї реалізації поданий на рисунку 4.15.

```
Python
import requests

# Отримати дані з веб-сервісу
response = requests.get("https://api.example.com/cart/items")

# Парсинг даних
data = response.json()

# Відображення списку товарів
for item in data:
    print(item["name"], item["price"])
```

Рисунок 4.15 – Приклад парсингу даних із веб-сервісу

У цьому прикладі використовується бібліотека requests для звернення до веб-сервісу та одержання даних у форматі JSON. Після цього використовується метод json() для парсингу даних

та одержання доступу до інформації про кожен товар. Цикл for відображенням списку товарів на екрані.

Вихід цього коду поданий на рисунку 4.16.

```
Product 1 | 100 USD
Product 2 | 200 USD
Product 3 | 300 USD
```

Рисунок 4.16 – Вихід коду

Тема 5. Проєктування баз даних під час інтеграції інформаційних систем

Лекція 8. Проєктування баз даних під час інтеграції інформаційних систем

Проєктування баз даних під час інтеграції інформаційних систем є складним завданням, яке потребує глибокого розуміння потреб користувачів, структури даних і механізмів обміну даними.

Першим кроком у проєктуванні бази даних для інтеграції інформаційних систем є *визначення потреб користувачів і стейкхолдерів*. Це є розумінням того, які дані потрібні користувачам, як ці дані будуть використовуватися і якою повинна бути їхня точність й актуальність. Також важливо визначити будь-які обмеження, які можуть бути накладені на базу даних, наприклад, бюджет, терміни або технічні вимоги.

Питання, які потрібно розглянути під час визначення потреб користувачів і стейкхолдерів можуть бути такими:

- Які дані потрібні користувачам?
- Як ці дані будуть використовуватися?
- Якою повинна бути точність і актуальність даних?
- Які обмеження можуть бути накладені на базу даних?

Після того, як потреби користувачів та стейкхолдерів були визначені, вони можуть бути документовані у формі специфікації вимог. **Специфікація вимог** – це документ, який описує потреби користувачів і стейкхолдерів у проєкті UML. Він повинен бути чітким і зрозумілим, щоб розробники могли зрозуміти, що хочуть користувачі та стейкхолдери.

UML (Unified Modeling Language) – це мова моделювання, яка використовується для візуалізації, описування та документування систем. UML можна використовувати для визначення потреб користувачів та стейкхолдерів за допомогою таких діаграм:

- *Діаграма використання (Use Case Diagram)* – використовується для опису взаємодії між користувачами та системою.
- *Діаграма сценарію (Scenario Diagram)* – деталізує окремий сценарій використання, показаний на діаграмі використання.
- *Діаграма класів (Class Diagram)* – використовується для опису структури системи, враховуючи її класи, атрибути та операції.

Діаграми використання та сценарії корисні для розуміння потреб користувачів і стейкхолдерів. Вони показують, як користувачі будуть взаємодіяти з системою та які дії вони будуть виконувати. Діаграма класів корисна для розуміння даних, які будуть зберігатися в базі даних.

Діаграма прецедентів складається з двох основних елементів:

- Прецеденти – це функції, які система повинна виконувати.
- Актори – це користувачі або інші системи, які взаємодіють із системою.

Прецеденти можна класифікувати за типом:

- Істотні прецеденти – це прецеденти, які є критичними для функціональності системи.
- Неістотні прецеденти – це прецеденти, які не є критичними для функціональності системи, але можуть бути корисними.

Прецеденти також можна класифікувати за типом:

- Системні прецеденти – це прецеденти, які стосуються взаємодії між системою та користувачами.
- Користувачі прецеденти – це прецеденти, які стосуються взаємодії між користувачами та системою.

Наступним кроком є **розуміння структури даних**, які використовуються в різних джерелах. Це містить аналіз форматів даних, їхніх взаємозв'язків і способів зберігання.

Також важливо визначити будь-які дублювання даних, які можуть існувати в різних джерелах.

На основі розуміння потреб користувачів і структури даних можна розпочати **оптимізацію схеми бази даних**. Це містить визначення оптимальної структури для зберігання даних, мінімізацію дублювання та забезпечення відповідності потребам користувачів.

Концептуальна модель є високорівневим описом бази даних, який не залежить від конкретного програмного забезпечення або обладнання. Вона використовується для визначення основних даних, які будуть зберігатися в базі даних, і їхніх взаємозв'язків.

Концептуальну модель можна описати за допомогою різних нотацій, таких як нотація UML або нотація ERD (Entity-Relationship Diagram). Нотація UML є більш загальною і може використовуватися для опису різних типів систем. Нотація ERD є більш специфічною для баз даних і використовується для опису даних і взаємозв'язків між ними.

Концептуальна модель складається з двох основних елементів:

- Типові сутності – це основні дані, які будуть зберігатися в базі даних. Типові сутності можуть бути фізичними об'єктами, такими як клієнти, продукти або співробітники, або абстрактними поняттями, такими як замовлення або продажі.
- Взаємозв'язки між типами сутностей – це зв'язки між типами сутностей. Взаємозв'язки можуть бути один до одного, один до багатьох або багато до багатьох.

Щоб створити концептуальну модель, необхідно виконати такі кроки:

1. Описати основні дані, які будуть зберігатися в базі даних. Це можна зробити, використовуючи методологію або нотацію, таку як Entity-Relationship (ER)-діаграма.
2. Описати взаємозв'язки між типами сутностей. Це також можна зробити, використовуючи ER-діаграми.

3. Перевірити концептуальну модель із користувачами та стейкхолдерами, щоб переконатися, що вона відповідає їхнім потребам.

Дублювання даних може бути серйозною проблемою для баз даних для інтеграції інформаційних систем. До його виникнення призводить: порушення правил синхронізації даних: неправильний дизайн бази даних: зміна бізнес-процесів.

Дублювання даних так само може призвести до таких проблем:

- Зниження продуктивності: дублювання даних може призвести до зниження продуктивності, оскільки базі даних потрібно буде обробляти більше даних.
- Складність управління: дублювання даних може ускладнити управління базою даних, оскільки потрібно буде проводити облік кількох копій одних і тих самих даних.
- Проблеми з безпекою: дублювання даних може призвести до проблем із безпекою, оскільки може бути складно відстежити зміни в даних.

Тому важливо **мінімізувати дублювання** даних у базах даних. Для цього можна використовувати такі методи:

- Правильний дизайн бази даних: Дублюванню даних можна запобігти за допомогою правильного дизайну бази даних. Наприклад, замість зберігання одних і тих самих даних в декількох таблицях, можна використовувати ключі для зв'язування даних між таблицями.
- Правила синхронізації даних: Дублюванню даних можна запобігти за допомогою правил синхронізації даних. Правила синхронізації даних забезпечують, щоб дані в різних системах були синхронізовані.
- Автоматизовані процеси: Дублюванню даних можна запобігти за допомогою автоматизованих процесів. Автоматизовані процеси можуть бути використані для перевірки даних на наявність дублікатів і для видалення дублікатів.

У базах даних для інтеграції інформаційних систем важливо мінімізувати дублювання даних, оскільки це може ускладнити інтеграцію систем. Для цього можна використовувати комбінацію вищезазначених методів.

Після того, як база даних була оптимізована, необхідно визначити **механізми та протоколи для обміну даними** між базами даних та іншими інформаційними системами. Це містить визначення способу передавання даних, їхнього формату та безпеки.

Існують різні протоколи обміну даними, такі як:

- Протоколи прикладного рівня – це протоколи, які використовуються програмними додатками для обміну даними. До прикладних протоколів прикладного рівня належать HTTP, FTP та SMTP.
- Протоколи транспортного рівня – це протоколи, які використовуються для передавання пакетів даних між комп'ютерами. До транспортних протоколів належать TCP та UDP.
- Протоколи мережевого рівня – це протоколи, які використовуються для маршрутизації пакетів даних через мережу. До мережевих протоколів належать IP та ARP.

Механізм обміну даними визначає, як дані будуть передаватися між базами даних та іншими інформаційними системами. До поширених механізмів передавання даних для обміну даними між базами даних та іншими інформаційними системами належать:

- Файли. Дані можуть бути обмінені між різними системами за допомогою файлів. Це простий механізм, який може бути використаний для обміну невеликими обсягами даних.
- API. API (Application Programming Interface) – це набір функцій, які дозволяють програмним системам взаємодіяти між собою. API можуть використовуватися для обміну даними між базами даних та іншими інформаційними системами.

- Протоколи передавання даних. Протоколи передавання даних – це правила, які визначають, як дані передаються між системами. Протоколи передавання даних можуть використовуватися для обміну даними між базами даних та іншими інформаційними системами в мережі.

Формат даних визначає, як дані будуть кодуватися для передавання. Існують різні формати даних, такі як JSON, XML та CSV.

Безпека даних визначає, як дані будуть захищені від несанкціонованого доступу, зміни або знищення. До заходів безпеки даних належать аутентифікація, авторизація та шифрування.

Під час вибору механізмів і протоколів обміну даними необхідно враховувати такі фактори:

- Типи даних, які будуть обмінюватися. Деякі механізми обміну даними краще підходять для певних типів даних. Наприклад, JSON добре підходить для обміну структурованими даними, а XML добре підходить для обміну неструктурованими даними.
- Розмір даних, які будуть обмінюватися. Деякі механізми обміну даними більш ефективні для передавання великих обсягів даних, ніж інші. Наприклад, TCP ефективніше для передавання великих обсягів даних, ніж UDP.
- Віддаленість між системами, які обмінюються даними. Деякі механізми обміну даними краще підходять для передавання даних на великі відстані, ніж інші. Наприклад, TCP ефективніше для передавання даних на великі відстані, ніж UDP.
- Безпечні вимоги. Деякі механізми обміну даними краще захищають дані від несанкціонованого доступу, зміни або знищення, ніж інші. Наприклад, TLS (Transport Layer Security) забезпечує більш високий рівень безпеки, ніж SSL (Secure Sockets Layer).

Після того, як механізми та протоколи обміну даними були визначені, необхідно розробити або адаптувати програмне

забезпечення для реалізації цих механізмів і протоколів. Це програмне забезпечення може бути розроблено всередині компанії або закуплено в сторонньої компанії.

Маппінг даних є процесом, який дозволяє пов'язати дані з різних джерел у єдиній базі даних, що є необхідним для забезпечення їхньої узгодженості та цілісності.

Маппінг даних має такі етапи:

1. Визначення даних, які повинні бути зіставлені. Необхідно визначити, які дані з різних джерел повинні бути зіставлені. Це можна зробити, вивчивши концептуальні моделі даних різних систем.
2. Опис атрибутів даних, які повинні бути зіставлені. Необхідно описати атрибути даних, які повинні бути зіставлені. Це містить визначення типу даних, довжини та формату атрибутів.
3. Визначення правил відповідності даних. Необхідно визначити правила, які визначать, як дані з різних джерел будуть відповідати один одному. Ці правила можуть бути простими, такими як порівняння значень атрибутів, або більш складними, такими як застосування алгоритмів оброблення даних.
4. Виконання маппінгу даних. Необхідно реалізувати правила відповідності даних, щоб пов'язати дані з різних джерел у єдиній базі даних. Це може бути здійснено за допомогою спеціальних інструментів або вбудованих функцій бази даних.

Маппінг даних може бути складним завданням, оскільки може бути важко знайти точні відповідності між даними з різних джерел. Однак він є важливим етапом у проєктуванні бази даних для інтеграції інформаційних систем, оскільки дозволяє забезпечити узгодженість і цілісність даних.

Існує кілька різних типів маппінгу даних:

- Фізичний маппінг. Фізичний маппінг визначає, як дані з різних джерел будуть фізично зберігатися в єдиній базі даних.

- Логічний маппінг. Логічний маппінг визначає, як дані з різних джерел будуть логічно пов'язані один з одним.
- Семантичний маппінг. Семантичний маппінг визначає, як дані з різних джерел будуть семантично пов'язані один з одним.

Маппінг даних є важливим процесом для інтеграції інформаційних систем. Він дозволяє пов'язати дані з різних джерел у єдиній базі даних, що робить їх доступними для використання в єдиній системі.

Важливо визначити **рівні доступу до даних** для різних користувачів і систем. Це допоможе забезпечити захист даних і запобігти несанкціонованому доступу.

Під час визначення рівнів доступу до даних необхідно враховувати такі фактори:

- Типи даних, які повинні бути доступні. Деякі дані, такі як конфіденційні дані, повинні бути доступні лише обмеженому колу користувачів.
- Ролі користувачів. Необхідно визначити різні ролі користувачів і надати кожній ролі відповідний рівень доступу до даних.
- Системи, які повинні мати доступ до даних. Необхідно визначити, які системи повинні мати доступ до даних, і надати кожній системі відповідний рівень доступу до даних.

Існує кілька різних рівнів доступу до даних:

- Читати. Користувачі з рівнем доступу «Читати» можуть лише читати дані.
- Писати. Користувачі з рівнем доступу «Писати» можуть читати та оновлювати дані.
- Створювати. Користувачі з рівнем доступу «Створювати» можуть створювати нові дані.
- Видаляти. Користувачі з рівнем доступу «Видаляти» можуть видаляти існуючі дані.

Існує кілька різних способів визначити рівні доступу до даних:

- Ролі користувачів. Ролі користувачів – це набір дозволів, які надаються користувачеві. Користувач може мати одну або кілька ролей.
- Привілеї. Привілеї – це конкретні дозволи, які надаються користувачеві. Користувач може мати один або кілька привілеїв.
- Права доступу. Права доступу – це дозволи, які надаються користувачеві на певний об'єкт бази даних. Користувач може мати один або кілька прав доступу на об'єкт бази даних.

Після того, як база даних була розроблена, необхідно провести її **оптимізацію** для забезпечення максимальної продуктивності. Це містить визначення способів покращання часу доступу до даних, оброблення запитів і пропускної здатності.

Існує кілька різних способів оптимізувати продуктивність бази даних:

1. Використання оптимальної структури бази даних:

- **Нормалізація** – це процес розбиття бази даних на дрібніші таблиці, які пов'язані між собою. Це може допомогти покращити продуктивність бази даних, зробивши її більш ефективною для зберігання та доступу до даних.
- **Денормалізація** – це процес додавання дублікатів даних у базу даних. Це може допомогти покращити продуктивність запитів, зробивши їх більш ефективними для сортування та пошуку.

Приклад. *Припустимо, що існує база даних, яка зберігає інформацію про клієнтів. Нормалізація може допомогти вам покращити продуктивність, створивши окрему таблицю для адрес клієнтів. Це дозволить уникнути повторення адрес у таблиці клієнтів, що може покращити продуктивність запитів, які використовують адреси клієнтів.*

Припустимо, що існує база даних, яка зберігає інформацію про продажі. Денормалізація може допомогти покращити продуктивність, додавши поле «Код продукту» до

таблиці продажів. Це дозволить швидко знайти продажі певного продукту, просто переглянувши поле «Код продукту» в таблиці продажів.

2. Створення індексів на часто використовуваних полях:

- Індекси створюються на полях, які використовуються для ідентифікації записів у таблиці. Вони можуть значно покращити час доступу до даних, зробивши їх більш ефективними для сортування та пошуку.
- Індекси створюються на полях, які використовуються для сортування записів у таблиці. Вони можуть значно покращити час доступу до даних, зробивши їх більш ефективними для сортування.

Приклад. *Припустимо, що існує база даних, яка зберігає інформацію про співробітників. Індекс на поле «Ім'я» може допомогти швидко знайти співробітників за їхніми іменами.*

Припустимо, що існує база даних, яка зберігає інформацію про продажі. Індекс на поле «Дата продажу» може допомогти швидко знайти продажі за датою.

3. Використання оптимізатора запитів:

- Вибір оптимального способу виконання запиту. Оптимізатор запитів може вибирати різні способи виконання запиту. Він може вибрати спосіб, який є найбільш ефективним для конкретної бази даних і набору даних.
- Використання планування запитів. Планування запитів – це процес, який використовується для визначення найкращого способу виконання запиту. Це може допомогти покращити продуктивність запитів, використовуючи оптимальний спосіб виконання запиту.

Приклад. *Припустимо, що існує база даних, яка зберігає інформацію про клієнтів. Оптимізатор запитів може допомогти покращити продуктивність запиту, який шукає всіх клієнтів, чиї імена починаються з літери «А». Оптимізатор запитів може використовувати індекс на*

поле «Ім'я», щоб знайти клієнтів, чиї імена починаються з літери «А», швидше, ніж якщо б він не використовував індекс.

Припустимо, що існує база даних, яка зберігає інформацію про продажі. Оптимізатор запитів може допомогти покращити продуктивність запиту, який шукає всі продажі, здійснені в певному місяці. Оптимізатор запитів може використовувати індекс на поле «Дата продажу», щоб знайти продажі, здійснені в певному місяці, швидше, ніж якщо б він не використовував індекс.

4. Паралелізм:

- Паралельний доступ до даних – це процес, який дозволяє кільком користувачам або процесам одночасно одержувати доступ до бази даних. Це може значно покращити продуктивність під час оброблення великих обсягів даних.
- Паралельні обчислення – це процес, який дозволяє кільком процесорам одночасно обробляти запит. Це може значно покращити продуктивність під час оброблення складних запитів.

Приклад. *Припустимо, що існує база даних, яка зберігає інформацію про клієнтів і їхні замовлення. Можна використовувати паралелізм, щоб покращити продуктивність запиту, який шукає всіх клієнтів, які замовляли певний продукт. Можна розділити запит на кілька частин, кожна з яких відповідає за оброблення клієнтів із певного діапазону імен. Це дозволить знайти всіх клієнтів, які замовляли певний продукт, швидше, ніж якщо б ви обробляли їх послідовно.*

Міграція та синхронізація даних

Перед впровадженням нової бази даних необхідно виконати міграцію даних із існуючих систем. Це також включає визначення способу синхронізації даних між базою даних й іншими інформаційними системами.

Міграція даних – це процес перенесення даних із існуючих систем до нової бази даних. Це може бути складним завданням, оскільки дані з різних систем можуть мати різні формати, структури та значення.

Під час міграції даних необхідно враховувати такі фактори:

- Тип даних. Деякі дані, такі як текст і цифри, можуть бути легко перенесені, тоді як інші дані, такі як зображення та відео, можуть вимагати додаткових зусиль.
- Обсяг даних. Якщо дані є великими, необхідно використовувати спеціальні інструменти для міграції даних.
- Час, який необхідний для міграції даних. Міграція даних може зайняти тривалий час, тому важливо планувати її заздалегідь.
- Безпека даних. Необхідно забезпечити безпеку даних під час міграції.

Існує кілька різних способів виконати міграцію даних:

- Вручну. Міграція даних може бути виконана вручну, але це може бути трудомістким і ризикованим завданням.
- За допомогою інструментів міграції даних. Інструменти міграції даних можуть допомогти автоматизувати процес міграції даних, що може зробити його більш ефективним і надійним.
- За допомогою послуг постачальника міграції даних. Постачальники міграції даних можуть допомогти вам виконати міграцію даних, використовуючи досвід й експертизу своїх фахівців.

Синхронізація даних – це процес забезпечення того, щоб дані в різних інформаційних системах були в актуальному стані. Це може бути складним завданням, оскільки дані можуть змінюватися в різних системах у різний час.

Під час синхронізації даних необхідно враховувати такі фактори:

- Частота синхронізації. Частота синхронізації залежить від того, як часто дані змінюються в різних системах.
- Спосіб синхронізації. Існує кілька різних способів синхронізувати дані, наприклад, за допомогою копіювання даних, оновлення даних або використання шлюзу даних.

- Безпека даних. Необхідно забезпечити безпеку даних під час синхронізації.

Існує кілька різних способів виконати синхронізацію даних:

- Вручну. Синхронізація даних може бути виконана вручну, але це може бути трудомістким і ризикованим завданням.
- За допомогою інструментів синхронізації даних. Інструменти синхронізації даних можуть допомогти автоматизувати процес синхронізації даних, що може зробити його більш ефективним і надійним.
- За допомогою послуг постачальника синхронізації даних. Постачальники синхронізації даних можуть допомогти вам виконати синхронізацію даних, використовуючи досвід і експертизу своїх фахівців.

Тема 6. Сервісна архітектура додатків. Веб-сервіси

Лекція 9. Сервісна архітектура додатків

Веб-сервіси – це спосіб взаємодії між різними програмами в Інтернеті. Вони використовують стандартні протоколи передавання даних, такі як HTTP, для обміну повідомленнями.

Веб-сервіс – це програмна система, яка пропонує свої послуги іншим програмам через мережу. Вона має чітко визначений інтерфейс і протокол повідомлень, які дозволяють іншим програмам взаємодіяти з нею.

Веб-сервіси можуть бути реалізовані будь-якою мовою програмування і на будь-якій платформі. Вони не залежать від мови програмування, операційної системи або пристрою. Веб-сервіси широко використовуються в бізнесі та в Інтернеті. Вони використовуються для інтеграції різних систем, для надання послуг клієнтам і для розширення функціональності додатків.

Ось деякі з *переваг використання веб-сервісів*:

- Інтероперабельність. Веб-сервіси дозволяють різним програмам, написаним на різних мовах програмування і працюючим на різних платформах, взаємодіяти один з одним.
- Експансивність. Веб-сервіси можна легко розширювати, додаючи нові послуги або параметри.
- Підтримка. Веб-сервіси широко підтримуються різними компаніями й організаціями.

Використання веб-сервісів

- Веб-сервіси можна використовувати для обміну даними між мобільними додатками і веб-сайтами. Це дозволяє користувачам синхронізувати свої дані між різними пристроями.
- Веб-сервіси можна використовувати для інтеграції систем управління бізнес-процесами (ERP) з веб-сайтами. Це дозволяє клієнтам отримувати доступ до інформації з ERP-систем через веб-сайт.
- Веб-сервіси можна використовувати для розширення функціональності веб-сайтів. Наприклад, веб-сервіс

можна використовувати для надання користувачам можливості бронювати номери в готелях або купувати товари в магазинах.

Розробники концепції веб-сервісів пропонують такі **сценарії застосування веб-сервісів:**

- *Веб-сервіси як реалізація логіки програми* (бізнес-логіки). Тобто, створення нової прикладної програми бізнес-логіка, якої реалізується у веб-сервісі (рис. 6.1).

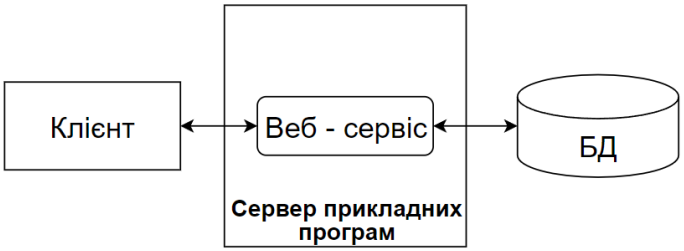

Рисунок 6.1 – Веб-сервіси як реалізація логіки програми

- *Веб-сервіси як засіб інтеграції.* Тобто, використання веб-сервісу як способу доступу віддалених клієнтів до внутрішньої ІС компанії, або для організації взаємодії компонента (наприклад, EJB, COM-компонента) з різними віддаленими клієнтами (рис. 6.2).

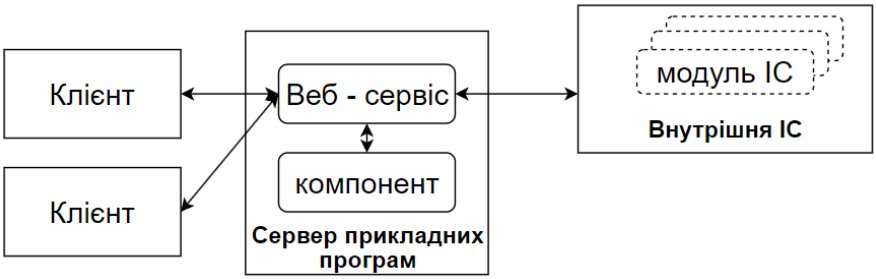

Рисунок 6.2 – Веб-сервіси як засіб інтеграції

- *Створення альтернативного сервісу.* У цьому разі під час розроблення нового веб-сервісу використовується опис інтерфейсу вже існуючого веб-сервісу. Отже, сервіс має багато потенційних клієнтів відразу з моменту початку роботи, а підключення до нього не вимагає істотних змін на стороні клієнта.

Концепція веб-сервісів має можливість ведення **реєстру веб-сервісів**. Опис інтерфейсу може бути одержано з такого реєстру. Після створення й впровадження нового веб-сервісу потрібно зареєструвати його в реєстрі. Тоді клієнти під час пошуку сервісів, що реалізують вихідний інтерфейс, отримають дані й на новий веб-сервіс.

- *Використання веб-сервісу як будівельного блоку під час створення програми.* Програма може використовувати веб-сервіси як вилучені компоненти, які надають певну функціональність. Існують різні сервіси, які надають якісне рішення таких задач як аутентифікація, ведення календаря, відправлення повідомлень, пошук, переклад і т. п. (рис. 6.3).

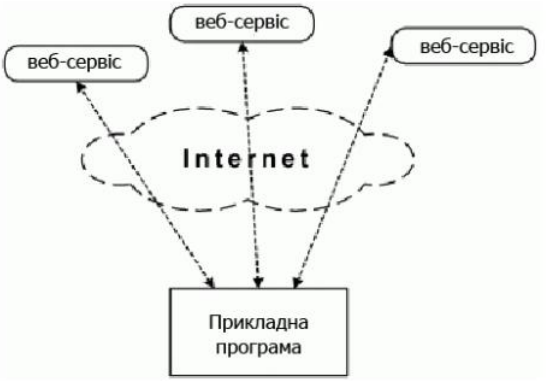

Рисунок 6.3 – Використання веб-сервісу як будівельного блоку

Існують логічні яруси мікросервісної архітектури

- *Клієнтська програма.* Цей ярус відповідає за взаємодію з користувачем. Він може містити веб-браузер, мобільний додаток або іншу програму, яка використовує мікросервісну систему.
- *Back-End.* Цей ярус містить мікросервіси, які відповідають за бізнес-логіку та оброблення даних. Він може мати такі сервіси, як:
- *API Gateway.* Цей сервіс забезпечує єдину точку входу для всіх мікросервісів. Він відповідає за маршрутизацію запитів до відповідних сервісів.
- *Experience мікросервіси.* Цей тип мікросервісів відповідає за забезпечення користувацького досвіду. Він може містити такі сервіси, як: логін і реєстрація; пошук; корзина покупок.
- *Доменні мікросервіси.* Цей тип мікросервісів відповідає за бізнес-логіку. Він може мати такі сервіси, як: управління продуктами; управління клієнтами; оплата.
- Data & *Integration*. Цей ярус містить такі компоненти, як: бази даних – компонент, що зберігає дані, які використовуються мікросервісами; інтеграційні інструменти, що використовуються для обміну даними між мікросервісами та іншими системами.

У мікросервісній архітектурі **міжсервісна комунікація** є основним фактором. Вона може бути синхронною або асинхронною, блокуючою або неблокуючою.

Синхронна комунікація означає, що сервіс очікує відповіді від іншого сервісу, перш ніж продовжити оброблення. **Асинхронна комунікація** означає, що сервіс не очікує відповіді від іншого сервісу, перш ніж продовжити оброблення.

Блокуюча комунікація означає, що сервіс блокується, поки не отримає відповідь від іншого сервісу. **Неблокуюча комунікація** означає, що сервіс не блокується, поки не отримає відповідь від іншого сервісу.

Для забезпечення відмовостійкості в мікросервісній архітектурі використовуються такі механізми, як тайм-аути,

повторні спроби, перемикачі ланцюгів, дедлайни та лімітери частоти.

Для забезпечення консистентності даних у мікросервісній архітектурі використовуються такі механізми, як подієво-орієнтована архітектура, транзакції в бізнес-рівні, події та шина подій.

API в мікросервісній архітектурі

API (Application Programming Interface) є набором програмних інтерфейсів, що дозволяють програмним компонентам взаємодіяти один з одним. В мікросервісній архітектурі API відіграють важливу роль, оскільки вони забезпечують взаємодію між окремими мікросервісами.

Існують два основних *типи API* в мікросервісній архітектурі:

- *Public API* – доступний для зовнішніх користувачів, наприклад, клієнтських додатків або сторонніх сервісів.
- *Internal API* – доступний лише для внутрішніх компонентів системи, наприклад, для інших мікросервісів або інфраструктурних компонентів.

API в мікросервісній архітектурі відіграють важливу роль з таких причин:

1. Вони дозволяють абстрагувати внутрішній склад мікросервісів від зовнішніх користувачів. Це полегшує розуміння та використання мікросервісів, а також їхню взаємодію з іншими системами.
2. Вони забезпечують стандартизований спосіб взаємодії між мікросервісами. Це полегшує розроблення і обслуговування мікросервісної системи.
3. Вони дозволяють масштабувати мікросервісну систему. API можна використовувати для ізоляції мікросервісів, що полегшує їхнє масштабування незалежно один від одного.

Для того щоб API в мікросервісній архітектурі були успішними, необхідно дотримуватися таких основних факторів:

1. API повинні бути добре документовані. Це полегшить їхнє розуміння та використання.

2. API повинні бути зручними у використанні. Вони повинні бути простими й інтуїтивно зрозумілими.

3. API повинні бути надійними. Вони повинні забезпечувати високий рівень доступності та продуктивності.

4. API повинні бути безпечними. Вони повинні захищати систему від несанкціонованого доступу.

Загальні рекомендації для розроблення API в мікросервісній архітектурі:

- Використовуйте стандартизовані протоколи і формати даних. Це полегшить їхню взаємодію з іншими системами.
- Використовуйте інструменти для автоматизації розроблення й тестування API. Це допоможе підвищити їхню якість і надійність.
- Використовуйте систему керування API для управління їхнім життєвим циклом. Це допоможе забезпечити їхнє ефективне розроблення, підтримку та обслуговування.

Мікросервісна **архітектура** може стати дуже складною, тому керування інфраструктурою та застосування DevOps best practices є важливими на всіх етапах розроблення. Розроблення масштабованого (scalable) та високодоступного (high-available) мікросервісного рішення може потребувати багато зусиль. Зокрема складно підтримувати систему, де є сотні мікросервісів.

Одним зі шляхів подолання такої ситуації є застосування домено-орієнтованого декаплінгу, який базується на групуванні мікросервісів по бізнес-домену (контексту) (рис. 6.4).

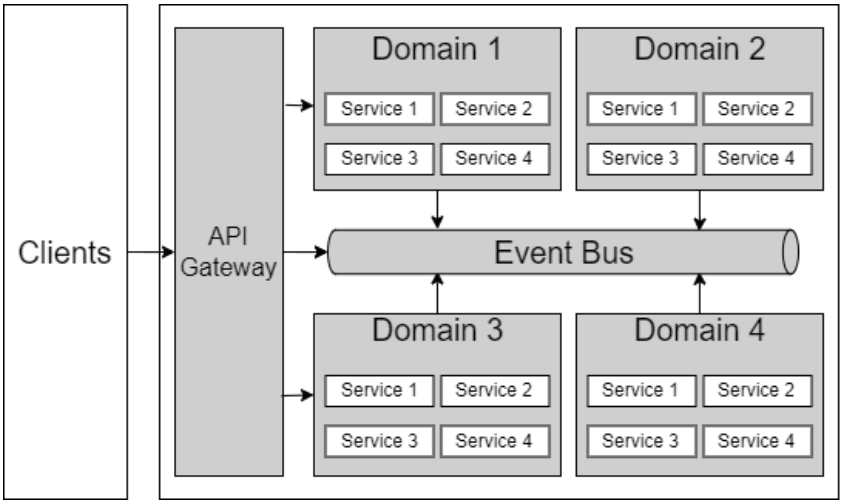

Рисунок 6.4 – Відокремлення на основі домену, кероване подіями

Такі системи поділяються на менші ділові домени (контексти), які взаємодіють через шину подій. Усередині кожного домену (контексту) сервіси можуть взаємодіяти між собою або через API, або через шину подій. Проте сервіси не мають прямого доступу до API сервісів інших доменів, вони можуть спілкуватися з ними лише через шину подій. Як результат, побудовані підсистеми менше за розміром і є більш керованими в сенсі підтримання та обслуговування.

Стандарти й технології

Робота веб-сервісів побудована на використанні декількох відкритих стандартів і технологій:

– XML – розширювана мова розмітки, призначена для зберігання й передавання структурованих даних;

– SOAP – протокол обміну повідомленнями на базі XML;

– WSDL – мова опису зовнішніх інтерфейсів веб-сервісів на базі XML;

– UDDI – універсальний інтерфейс розпізнавання, опису та інтеграції (Universal Discovery, Description, and Integration).

Каталог вебсервісів і відомостей про компанії, що надають веб-сервіси в загальне користування або конкретним компаніям.

Взаємозв'язок цих технологій можна умовно подати так, як показано на рисунку 6.5.

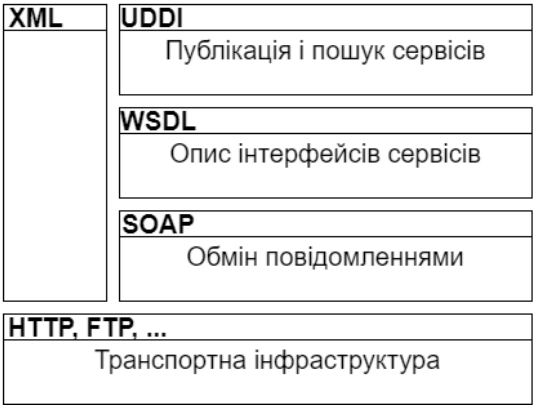

Рисунок 6.5 – Реалізація веб-сервісів

Веб-сервіси є одним із варіантів реалізації компонентної архітектури. Виділяють **два основних типи веб-сервісів** залежно від використовуваних технологій SOAP-сервіси та REST-сервіси.

Лекція 10. Веб-сервіси SOAP

Веб-сервіси SOAP – це вид веб-сервісів, які використовують протокол SOAP для обміну повідомленнями.

SOAP-сервіси, засновані на стандартах для веб-сервісів, розроблених міжнародним консорціумом W3C. Цей вид сервісів складніший у розробленні й використанні, тому застосовується в тому разі, коли важливе дотримання стандартів і високий ступінь формалізації. Одна з основних галузей застосування SOAP-сервісів – комерційні (enterprise) системи.

SOAP-сервіси використовують два базових стандарти, заснованих на XML:

- SOAP – стандарт передачі даних поверх високорівневих протоколів (наприклад, HTTP). Крім даних веб-сервісів можуть передаватися заголовки, використовувані, наприклад, для аутентифікації або для опису пріоритету запитів.
- WSDL – формат опису інтерфейсу й реалізації веб-сервісів. Інтерфейс веб-сервісу являє собою набір операцій (аналог методів для об'єктів), для кожної з яких визначена структура вхідних і вихідних повідомлень. Реалізація сервісу є ще однією або більше прив'язок (binding), які визначають спосіб виклику операцій.

Крім цих стандартів, для веб-сервісів визначені допоміжні технології, такі як безпека даних. Для інтеграції SOAP-сервісів розроблена мова WS-BPEL (теж заснована на XML), що дозволяє запитувати і обробляти дані для довільного числа сервісів. Засоби для реалізації і використання сервісів входять, наприклад, в Java EE і Microsoft .NET.

SOAP-повідомлення складається з заголовка і тіла.

Заголовок SOAP-повідомлення складається з таких елементів:

- **Envelope.** Це основний елемент SOAP-повідомлення. Він містить усі інші елементи заголовка та тіла.

- **Header.** Це необов'язковий елемент, який містить додаткову інформацію про повідомлення, наприклад, інформацію про безпеку або надійність.
- **Body.** Це основний елемент SOAP-повідомлення, який містить корисні дані.

Тіло SOAP-повідомлення складається з таких елементів:

- **Message**. Це основний елемент тіла SOAP-повідомлення. Він містить дані, які потрібно передати.
- **Fault**. Це необов'язковий елемент, який містить інформацію про помилку, яка сталася під час оброблення повідомлення.

На рисунку 6.6 подано SOAP-повідомлення. Що містить запит на одержання поточної дати й часу.

```
XML
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header/>
  <soap:Body>
    <GetDateTime xmlns="http://example.com/soap">
    </GetDateTime>
  </soap:Body>
</soap:Envelope>
```

Рисунок 6.6 – Приклад SOAP-повідомлення

WSDL (Web Services Description Language) – це мова опису веб-сервісів. Вона використовується для визначення інтерфейсу веб-сервісу та його доступних операцій.

На рисунку 6.7 подано приклад WSDL-опису, що визначає веб-сервіс, який надає операцію для одержання поточної дати і часу.

```xml
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://example.com/soap"
             targetNamespace="http://example.com/soap">
  <portType name="DateTimeService">
    <operation name="GetDateTime">
      <input message="tns:GetDateTimeRequest"/>
      <output message="tns:GetDateTimeResponse"/>
    </operation>
  </portType>
  <message name="GetDateTimeRequest">
    <part name="parameters" element="tns:GetDateTimeParameters"/>
  </message>
  <message name="GetDateTimeResponse">
    <part name="parameters" element="tns:GetDateTimeParameters"/>
  </message>
  <types>
    <schema targetNamespace="http://example.com/soap">
      <element name="GetDateTimeParameters">
        <complexType>
          <sequence>
          </sequence>
        </complexType>
      </element>
    </schema>
  </types>
</definitions>
```

Рисунок 6.7 – Приклад WSDL-опису

У цьому WSDL-описі визначений порт DateTimeService, який реалізує операцію GetDateTime. Операція GetDateTime приймає один аргумент parameters, який є необов'язковим. Операція GetDateTime повертає один результат parameters, який також є необов'язковим.

Загалом WSDL-опис складається з таких елементів:

- **Опис порту** – основний елемент WSDL-опису. Він визначає порт веб-сервісу, який відповідає за певну операцію.
- **Опис операції** – елемент, який визначає операцію, яку реалізує порт.
- **Опис аргументів** – елементи, які визначають аргументи, які приймає операція.

- **Опис результатів** – елементи, які визначають результати, які повертає операція.

Веб-сервіси SOAP можуть бути розгорнуті на різних платформах, таких як Java, NET і Python. Для розгортання веб-сервісу SOAP необхідно створити серверний додаток, який реалізує протокол SOAP.

У загальному випадку процес розгортання веб-сервісу SOAP має такі кроки:

1. Розроблення серверного додатка. Серверний додаток реалізує операції, які надаються веб-сервісом.
2. Створення WSDL-опису. WSDL-опис визначає інтерфейс веб-сервісу та його доступні операції.
3. Розміщення серверного додатка. Серверний додаток розміщується на сервері, який доступний із мережі.
4. Публікування WSDL-опису. WSDL-опис публікується в каталозі веб-сервісів, наприклад, в UDDI.

Після того, як веб-сервіс буде розгорнуто, клієнтські додатки можуть взаємодіяти з ним, використовуючи WSDL-опис.

Приклад того, як розгорнути веб-сервіс SOAP на Java:

1. Створіть новий проєкт Java.
2. Додайте в проєкт пакет com.example.soap.
3. Створіть у пакеті клас DateTimeService.
4. У класі DateTimeService реалізуйте операцію GetDateTime.
5. Створіть WSDL-опис веб-сервісу.
6. Розмістіть серверний додаток на сервері.
7. Публікуйте WSDL-опис у каталозі веб-сервісів.

Для взаємодії з веб-сервісом SOAP необхідно використовувати клієнтський додаток, який також реалізує протокол SOAP. Клієнтський додаток використовується для створення SOAP-повідомлень, їхнього передавання веб-сервісу та оброблення отриманих відповідей. Клієнтський додаток може бути розроблено на будь-якій мові програмування, яка підтримує протокол SOAP.

Загальний алгоритм взаємодії клієнтського додатка з веб-сервісом SOAP:

1. Створення SOAP-повідомлення. Клієнтський додаток створює SOAP-повідомлення, яке містить запит на операцію, яку потрібно виконати на веб-сервісі.
2. Відправка SOAP-повідомлення. Клієнтський додаток відправляє SOAP-повідомлення веб-сервісу.
3. Одержання відповіді від веб-сервісу. Веб-сервіс обробляє запит і відправляє відповідь клієнтському додатку.
4. Оброблення відповіді від веб-сервісу. Клієнтський додаток обробляє одержану відповідь і отримує результат виконання операції.

Приклад того, як створити додаток клієнту на Java для взаємодії з веб-сервісом SOAP поданий на рисунку 6.8.

Цей клієнтський додаток створює запит на одержання поточної дати і часу, створює SOAP-повідомлення, додає запит до SOAP-повідомлення, відправляє SOAP-повідомлення веб-сервісу й одержує відповідь від веб-сервісу. Потім додаток виводить результат на консоль.

```
Java
import com.example.soap.DateTimeService;
import com.example.soap.GetDateTimeRequest;
import com.example.soap.GetDateTimeResponse;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
public class Main {
    public static void main(String[] args) throws JAXBException {
        // Створити запит на отримання поточної дати і часу
        GetDateTimeRequest request = new GetDateTimeRequest();
        // Створити SOAP-повідомлення
        MessageFactory messageFactory = MessageFactory.newInstance();
        SOAPMessage soapMessage = messageFactory.createMessage();
        SOAPPart soapPart = soapMessage.getSOAPPart();
        // Додати запит до SOAP-повідомлення
        soapPart.getEnvelope().getBody().addBodyElement(request.getQName());
        // Відправити SOAP-повідомлення
        soapMessage.send("http://localhost:8080/DateTimeService");
        // Отримати відповідь від веб-сервісу
        Unmarshaller unmarshaller = JAXBContext.newInstance(GetDateTimeResponse.class).createUnmarshaller
        GetDateTimeResponse response = (GetDateTimeResponse) unmarshaller.unmarshal(
            soapMessage.getSOAPBody().getTextContent()
        );
        // Вивести результат на консоль
        System.out.println(response.getDateTime());
    }
}
```

Рисунок 6.8 – Приклад додатка клієнта для взаємодії з веб-сервісом SOAP

Використання WS-Security для забезпечення безпеки повідомлень SOAP.

WS-Security – це стандарт, який додає підтримку безпеки до повідомлень SOAP. Він використовується для захисту повідомлень SOAP від несанкціонованого доступу, зміни або підроблення.

WS-Security реалізується за допомогою набору хеш-функцій, сигнатур і ключів. Веб-сервіси SOAP, які реалізують WS-Security, можуть використовувати ці методи для захисту повідомлень SOAP.

Існує декілька способів, як WS-Security може бути використаний для захисту повідомлень SOAP:

- **Хешування** може бути використано для створення контрольної суми повідомлень SOAP. Ця контрольна сума може потім використовуватися для перевірки цілісності повідомлення.

- **Сигнатура** може бути використана для підписання повідомлень SOAP. Цей підпис може потім використовуватися для перевірки автентичності відправника повідомлення.
- **Ключі** можуть використовуватися для шифрування повідомлень SOAP. Цей шифр може потім використовуватися для захисту повідомлень SOAP від несанкціонованого доступу.

Приклад того, як використовувати WS-Security для забезпечення безпеки повідомлень SOAP поданий на рисунку 6.9.

```
XML
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <soap:Header>
    <wsse:Security soap:mustUnderstand="1" xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.x
      <wsse:UsernameToken wsu:Id="UsernameToken-1">
        <wsse:Username>user1</wsse:Username>
        <wsse:Password>password1</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <GetDateTime xmlns="http://example.com/soap">
    </GetDateTime>
  </soap:Body>
</soap:Envelope>
```

Рисунок 6.9 – Приклад використання WS-Security

У цьому прикладі до SOAP-повідомлення додано заголовок Security, який містить інформацію про автентифікацію відправника. В цьому разі відправником є користувач user1, який зазначив пароль password1.

Використання WS-ReliableMessaging для забезпечення надійності доставки повідомлень

WS-ReliableMessaging – це стандарт, який додає підтримку надійності до повідомлень SOAP. Він використовується для гарантування того, що повідомлення SOAP будуть доставлені до одержувача.

WS-ReliableMessaging реалізується за допомогою набору механізмів, таких як підтвердження доставки, відновлення доставлення та відновлення передавання. Веб-сервіси SOAP, які реалізують WS-ReliableMessaging, можуть використовувати ці механізми для гарантування надійності доставлення повідомлень SOAP.

Існує декілька способів, як WS-ReliableMessaging може бути використаний для забезпечення надійності доставлення повідомлень SOAP:

- **Підтвердження доставлення** може бути використано для інформування відправника повідомлення про те, що повідомлення було успішно доставлено до одержувача.
- **Відновлення доставлення** може бути використано для повторного відправлення повідомлень SOAP, які не були успішно доставлені.
- **Відновлення передавання** може бути використано для відновлення передавання повідомлень SOAP, які були перервані.

Приклад того, як використовувати WS-ReliableMessaging для забезпечення надійності доставлення повідомлень SOAP поданий на рисунку 6.10.

У наведеному прикладі до SOAP-повідомлення додано заголовок SequenceAcknowledgement, який містить інформацію про статус доставлення повідомлення. У цьому разі повідомлення було успішно доставлено, і сервер підтвердив одержання повідомлення.

```
XML
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:wsrm="http://docs.oasis-open.org/ws-rx/wsrm/200702">
  <soap:Header>
    <wsrm:SequenceAcknowledgement soap:mustUnderstand="1" xmlns:wsrm="http://docs.oasis-open.org/ws-rx/wsrm/2007
      <wsrm:Identifier>123456</wsrm:Identifier>
      <wsrm:AcknowledgementRange>0-10</wsrm:AcknowledgementRange>
    </wsrm:SequenceAcknowledgement>
  </soap:Header>
  <soap:Body>
    <GetDateTime xmlns="http://example.com/soap">
    </GetDateTime>
  </soap:Body>
</soap:Envelope>
```

Рисунок 6.10 – Приклад використання WS-ReliableMessaging

Лекція 11. Веб-сервіси RESTFull

Архітектурний стиль REST (Representational State Transfer) – це спосіб проєктування й реалізації веб-сервісів. REST використовує стандартні протоколи й методи HTTP для обміну даними між клієнтом і сервером.

REST заснований на чотирьох основних принципах:

- **Стандартизація**. REST використовує стандартні протоколи й методи HTTP, що забезпечує його інтероперабельність. Це означає, що REST-сервіси можуть використовуватися з різними клієнтськими програмами, незалежно від того, на якій платформі вони працюють.
- **Сервіси як ресурси**. REST-сервіси є ресурсами. Кожен ресурс має унікальний URI, який використовується для його ідентифікації. URI повинен бути унікальним і легко запам'ятовуватися. Він може використовуватися для створення складних структур ресурсів.
- **Статичні відносини**. Ресурси взаємодіють один з одним за допомогою статичних відносин. Це означає, що клієнти не повинні знати про внутрішню структуру веб-сервісу. Це полегшує розроблення клієнтських програм і забезпечує більшу надійність.
- **Неявність стану**. REST-сервіси не зберігають стан клієнта. Це означає, що клієнт повинен передавати всю необхідну інформацію в кожному запиті. Це може призвести до більшої кількості запитів, але це також робить REST-сервіси більш масштабованими.

Ці принципи допомагають забезпечити простоту, інтероперабельність і масштабованість REST-сервісів.

RESTful веб-сервіси – це веб-сервіси, які використовують архітектуру REST (REpresentational State Transfer). RESTful веб-сервіси мають низку переваг перед іншими типами веб-сервісів, зокрема легкість у масштабуванні, тестуванні, документуванні, використанні різними програмними мовами та платформами.

RESTful веб-сервіси використовують HTTP для взаємодії з клієнтами. HTTP – це протокол прикладного рівня, який використовується для передавання інформації в Інтернеті.

RESTful web-services використовують HTTP-запити для виконання операцій. Кожному типу операції відповідає певний HTTP-запит:

- GET: Використовується для одержання ресурсу.
- POST: Використовується для створення нового ресурсу.
- PUT: Використовується для оновлення існуючого ресурсу.
- DELETE: Використовується для видалення ресурсу.

Приклади базових принципів проєктування REST-сервісів:

Явне використання HTTP-методів. HTTP-методи використовуються для визначення операцій, які можна виконувати на ресурсі. Наприклад, метод GET використовується для одержання ресурсу, метод POST використовується для створення ресурсу, метод PUT використовується для оновлення ресурсу, а метод DELETE використовується для видалення ресурсу.

Приклади:

- GET /products – одержати список усіх продуктів.
- POST /products – створити новий продукт.
- PUT /products/12345 – оновити продукт з ідентифікатором 12345.
- DELETE /products/12345 – видалити продукт з ідентифікатором 12345.

У прикладі на рисунку 6.11 показано, як використовувати HTTP-методи для взаємодії з REST-сервісом.

```
Python
import requests
# GET /products
response = requests.get("https://example.com/products")
products = response.json()
# POST /products
product = {"name": "New Product"}
response = requests.post("https://example.com/products", json=product)
# PUT /products/12345
product = {"name": "Updated Product"}
response = requests.put("https://example.com/products/12345", json=produc
# DELETE /products/12345
response = requests.delete("https://example.com/products/12345")
```

Рисунок 6.11 – Приклад явного використання HTTP-методів

У цьому прикладі використовується модуль requests для відправлення HTTP-запитів до REST-сервісу. Метод requests.get() використовується для одержання ресурсу, метод requests.post() використовується для створення ресурсу, метод requests.put() використовується для оновлення ресурсу, а метод requests.delete() використовується для видалення ресурсу.

Явне використання HTTP-методів допомагає зробити REST-сервіси більш зрозумілими і передбачуваними. Це також допомагає клієнтам REST-сервісів використовувати правильні методи для виконання необхідних операцій.

Незбереження стану. REST-сервіси не зберігають стан клієнта. Це означає, що кожний запит є самостійним і не залежить від попередніх запитів. Це дозволяє REST-сервісам масштабуватися й бути більш надійними. У програмному коді незбереження стану можна реалізувати різними способами. Одним із способів є *використання HTTP-заголовків* для передавання інформації між клієнтом і сервером. Наприклад, клієнт може використовувати HTTP-заголовок *Authorization* для передавання токена аутентифікації серверу. Сервер може використовувати цей токен для аутентифікації клієнта і дозволити йому одержати доступ до ресурсів. Іншим способом реалізації незбереження стану є використання *сеансів*. Сеанс –

це механізм, який дозволяє серверу зберігати інформацію про клієнта впродовж певного періоду часу. Це може бути корисно для таких речей, як зберігання вмісту кошика покупок або відстеження прогресу користувача у веб-додатку. Незбереження стану також можна реалізувати за допомогою *кешу*. Кеш – це механізм, який зберігає копії даних у пам'яті сервера. Це може бути корисно для прискорення доступу до часто використовуваних даних.

У прикладах на рисунку 6.12 показано реалізацію незбереження стану.

```
Python
# HTTP-заголовки
import requests
# Отримати токен аутентифікації
response = requests.post("https://example.com/oauth/token",
                         data={"username": "user", "password": "password"})
token = response.json()["access_token"]
# Використовувати токен аутентифікації для отримання доступу до ресурсу
response = requests.get("https://example.com/products",
                        headers={"Authorization": "Bearer " + token})
products = response.json()
# Сеанси
import requests
# Зареєструватися
response = requests.post("https://example.com/auth/register",
                        data={"username": "user", "password": "password"})

# Увійти
response = requests.post("https://example.com/auth/login",
                        data={"username": "user", "password": "password"})
# Зберегти товар у кошику
response = requests.post("https://example.com/cart/add",
                        headers={"Authorization": "Bearer " + response.json()["access_token"]
                        data={"product_id": 12345})
# Переглянути кошик
response = requests.get("https://example.com/cart",
                        headers={"Authorization": "Bearer " + response.json()["access_token"]
cart = response.json()
# Кеш
import requests
# Отримати список продуктів
response = requests.get("https://example.com/products")
products = response.json()
# Зберегти список продуктів у кеші
cache.set("products", products, timeout=60)
# Отримати список продуктів з кешу
products = cache.get("products")
```

Рисунок 6.12 – Приклад незбереження стану

Надання URI, аналогічних структурі каталогів

URI REST-сервісів повинні бути розроблені так, щоб вони відображали структуру ресурсів, які вони представляють. Наприклад, URI для ресурсу «продукт» може бути "/products/12345".

Це допомагає клієнтам REST-сервісів зрозуміти, що являє собою ресурс і які операції можна виконувати на цьому ресурсі.

Приклади URI, які відповідають структурі каталогів:

- GET /products – одержати список всіх продуктів.
- POST /products – створити новий продукт.
- PUT /products/12345 – оновити продукт з ідентифікатором 12345.
- DELETE /products/12345 – видалити продукт з ідентифікатором 12345.
- GET /users – одержати список всіх користувачів.
- POST /users – створити нового користувача.
- PUT /users/12345 – оновити користувача з ідентифікатором 12345.
- DELETE /users/12345 – видалити користувача з ідентифікатором 12345.
- GET /orders – одержати список усіх замовлень.
- POST /orders – створити нове замовлення.
- PUT /orders/12345 – оновити замовлення з ідентифікатором 12345.
- DELETE /orders/12345 – видалити замовлення з ідентифікатором 12345.

Ці приклади демонструють, як URI REST-сервісів можуть бути розроблені так, щоб вони відображали структуру ресурсів, які вони представляють.

У прикладі на рисунку 6.13 показано, як використовувати URI, які відповідають структурі каталогів, для взаємодії з REST-сервісом.

```
Python
import requests
# Отримати список всіх продуктів
response = requests.get("https://example.com/products")
products = response.json()
# Створити новий продукт
product = {"name": "New Product"}
response = requests.post("https://example.com/products", json=product)
# Оновити продукт з ідентифікатором 12345
product = {"name": "Updated Product"}
response = requests.put("https://example.com/products/12345", json=product)
# Видалити продукт з ідентифікатором 12345
response = requests.delete("https://example.com/products/12345")
```

Рисунок 6.13 – Приклад використання URI

У наведеному прикладі використовується модуль requests для відправлення HTTP-запитів до REST-сервісу. URI REST-сервісу відповідають структурі каталогів, що робить їх простими для розуміння.

Передавання даних у XML, JSON або в обох форматах. Дані, які передаються між клієнтом і REST-сервісом, можуть бути подані в різних форматах, таких як XML, JSON або SOAP. Найпоширенішим форматом для REST-сервісів є JSON.

REST-сервіси можуть використовувати один формат даних або обидва формати даних. Вибір формату даних залежить від конкретних потреб REST-сервісу.

У прикладі на рисунку 6.14 показано, як використовувати XML для передавання даних до REST-сервісу.

```
Python
import requests
# Створити об'єкт XML
product = {"name": "New Product"}
product_xml = "<product><name>New Product</name></product>"
# Відправити запит до REST-сервісу
response = requests.post("https://example.com/products", data=product_xm
# Отримати відповідь від REST-сервісу
response_json = response.json()
```

Рисунок 6.14 – Приклад використання XML для передавання даних до REST-сервісу

У цьому прикладі використовується модуль requests для відправлення HTTP-запиту до REST-сервісу. Дані в XML-форматі передаються як параметр data.

У прикладі на рисунку 6.15 показано, як використовувати JSON для передаванні даних до REST-сервісу.

```
Python
import requests
# Створити об'єкт JSON
product = {"name": "New Product"}
# Відправити запит до REST-сервісу
response = requests.post("https://example.com/products", json=produc
# Отримати відповідь від REST-сервісу
response_json = response.json()
```

Рисунок 6.15 – Приклад використання JSON для передавання даних до REST-сервісу

У цьому прикладі використовується модуль requests для відправлення HTTP-запиту до REST-сервісу. Дані в JSON-форматі передаються як параметр json.

Для **забезпечення надійності доставлення повідомлень** RESTful застосовують такі методи:

- *Повторні спроби доставлення*. Якщо повідомлення не було доставлено з першого разу, сервер RESTful може зробити повторну спробу доставки повідомлення. Це може допомогти забезпечити, що повідомлення буде доставлено, навіть якщо виникли тимчасові проблеми з мережею. Наприклад, RESTful веб-сервіс, який надає клієнтам доступ до інформації про погоду, може використовувати повторні спроби доставлення, щоб гарантувати, що клієнти одержують найактуальнішу інформацію. Якщо перша спроба доставлення повідомлення про погоду не вдалася, сервер RESTful може зробити повторну спробу доставлення повідомлення через кілька секунд або хвилин.
- *Асинхронна доставка.* Під час асинхронного доставлення повідомлення не потрібно чекати, поки повідомлення буде доставлено. Це може допомогти забезпечити,

що клієнт не буде заблоковано, якщо повідомлення не було доставлено. Наприклад, RESTful веб-сервіс, який дозволяє клієнтам замовляти товари, може використовувати асинхронне доставлення, щоб клієнти могли продовжувати додавати товари в кошик, навіть якщо сервер не зміг доставити повідомлення про замовлення одразу.

- *Алерти про помилки.* Якщо повідомлення не може бути доставлено, сервер RESTful може надіслати клієнту попередження про помилку. Це допоможе клієнту дізнатися, що повідомлення не було доставлено, і вжити заходів для вирішення проблеми. Наприклад, RESTful веб-сервіс, який дозволяє клієнтам відправляти електронні листи, може надіслати клієнту попередження про помилку, якщо електронний лист не був доставлений адресату.

Для **забезпечення безпеки повідомлень** RESTful застосовують такі методи:

- *Шифрування* використовується для захисту повідомлень від несанкціонованого доступу. Наприклад, RESTful веб-сервіс, який надає клієнтам доступ до конфіденційних даних, може використовувати шифрування, щоб захистити ці дані від несанкціонованого доступу.
- *Аутентифікація* використовується для перевірки особи відправника повідомлення. Наприклад, RESTful веб-сервіс, який дозволяє клієнтам здійснювати платежі, може використовувати аутентифікацію, щоб переконатися, що платіж здійснюється дійсним клієнтом.
- *Авторизація* використовується для визначення того, чи має відправник повідомлення – доступ до ресурсу, на який він посилається. Наприклад, RESTful веб-сервіс, який надає клієнтам доступ до різних ресурсів, може використовувати авторизацію, щоб переконатися, що клієнт має доступ лише до тих ресурсів, які він має право використовувати.

На додаток до цих методів, для забезпечення надійності та безпеки повідомлень RESTful також можуть використовуватися такі технології, як:

- *MQTT* – це протокол передавання повідомлень, який розроблений для забезпечення надійності та ефективності передавання повідомлень в умовах обмежених ресурсів.
- *AMQP* – це протокол передавання повідомлень, який розроблений для забезпечення надійності та масштабованості передавання повідомлень.
- *Kafka* – це платформа потокового оброблення даних, яка може використовуватися для зберігання та оброблення повідомлень у реальному часі.

Лекція 12. Створення та захист RESTful APIs

API, або програмний інтерфейс додатків, – це набір функцій, які дозволяють програмним продуктам взаємодіяти один з одним. API описує, як програми можуть запитувати дані та виконувати дії на іншому програмному продукті (рис. 6.16).

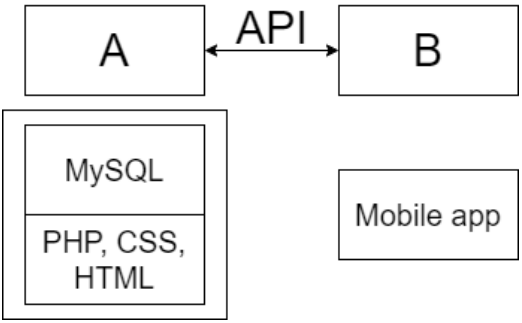

Рисунок 6.16 – Приклад роботи API

API можна використовувати для багатьох різних цілей, таких як: обміну даними між різними програмними продуктами; автоматизації завдань; розширення можливостей програм.

Існує два основних **типи API**:

- *Протокольні* API використовують стандартний протокол, наприклад HTTP, для передачі даних. REST API є прикладом протокольного API.
- *Спеціалізовані* API використовують власний протокол, який може бути більш ефективним або зручним для конкретного застосування.

REST API (Representational State Transfer API) – це тип протокольного API, який використовує HTTP для передачі даних.

Діаграма на рисунку 6.17 показує загальну модель REST API. Під час переходу на веб-сайт за допомогою браузера надсилається запит до веб-сервера за протоколом HTTP. Веб-сервер відповідає на запит, надсилаючи контент веб-сторінки. Браузер відображає контент веб-сторінки.

Цей процес обміну повідомленнями між браузером і веб-сервером є прикладом роботи протоколу HTTP. HTTP – це протокол прикладного рівня, який використовується для передачі даних через Інтернет. Він працює за моделлю клієнт – сервер, де браузер є клієнтом, а веб-сервер – сервером.

HTTP використовує запити і відповіді для обміну повідомленнями між клієнтом і сервером. Запит – це повідомлення, яке відправляє клієнт серверу. Відповідь – це повідомлення, яке відправляє сервер клієнту.

Цей шаблон запиту та відповіді є принципом роботи API REST.

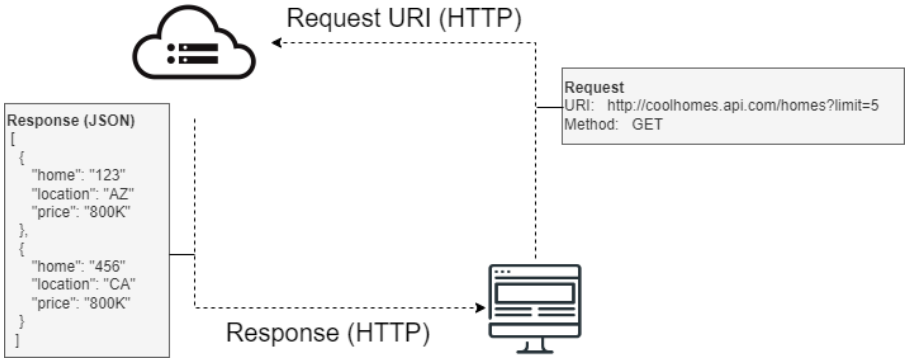

Рисунок 6.17 – Приклад загальної моделі REST API

Серіалізація та десеріалізація – це два важливі процеси в RESTful APIs. *Серіалізація* – це процес перетворення об’єктів у формат, який можна передавати через мережу. *Десеріалізація* – це процес перетворення даних, одержаних через мережу, назад у об’єкти.

RESTful APIs використовують стандартні формати даних, такі як JSON або XML для передавання інформації між клієнтом і сервером. JSON є більш популярним вибором, оскільки він більш компактний і легше читається.

Для серіалізації об’єктів у JSON можна використовувати стандартні функції мов програмування. У прикладі на рисунку

6.18 показано, як у Python можна використовувати функцію json.dumps().

```
Python
import json
# Create an object
user = {
    "username": "johndoe",
    "email": "johndoe@example.com",
    "password": "password123"
}
# Serialize the object to JSON
json_data = json.dumps(user)
# Print the JSON data
print(json_data)

_______
Вихід:
JSON
{"username": "johndoe", "email": "johndoe@example.com", "password": "password123"}
```

Рисунок 6.18 – Приклад серіалізації

Для десеріалізації даних JSON в об’єкти можна використовувати стандартні функції мов програмування. У прикладі на рисунку 6.19 показано, як у Python можна використовувати функцію json.loads().

```
Python
import json
# Create JSON data
json_data = """
{
    "username": "johndoe",
    "email": "johndoe@example.com",
    "password": "password123"
}
"""
# Deserialize the JSON data to an object
user = json.loads(json_data)

# Print the object
print(user)


_____
Вихід:
Python
{'username': 'johndoe', 'email': 'johndoe@example.com', 'password': 'password123
```

Рисунок 6.19 – Приклад десеріалізації

Приклади використання серіалізації та десеріалізації в RESTful APIs

- *Запит на одержання об'єкта*. Клієнт може створити JSON-дані з об'єкта, який він хоче отримати, а потім відправити ці дані на сервер за допомогою HTTP-запиту GET. Сервер може використовувати десеріалізацію, щоб отримати об'єкт з одержаних JSON-даних, а потім повернути цей об'єкт клієнту.
- *Запит на створення об'єкта.* Клієнт може створити JSON-дані з новим об'єктом, який він хоче створити, а потім відправити ці дані на сервер за допомогою HTTP-запиту POST. Сервер може використовувати десеріалізацію, щоб отримати об'єкт з одержаних JSON-даних, а потім створити цей об'єкт у базі даних.
- *Запит на оновлення об'єкта*. Клієнт може створити JSON-дані з об'єктом, який він хоче оновити, а потім відправити ці дані на сервер за допомогою HTTP-запиту PUT. Сервер може використовувати десеріалізацію, щоб отримати об'єкт з одержаних JSON-даних, а потім оновити цей об'єкт у базі даних.
- *Запит на видалення об'єкта.* Клієнт може відправити HTTP-запит DELETE на сервер для видалення об'єкта. Сервер може використовувати десеріалізацію, щоб отримати об'єкт з одержаних JSON-даних, а потім видалити цей об'єкт у базі даних.

Версіювання RESTful API – це процес, який дозволяє клієнтам і серверам взаємодіяти з різними версіями API. Це необхідно для того, щоб клієнти могли продовжувати використовувати API, навіть якщо на сервері відбулися зміни.

Є кілька різних способів версіонування RESTful API. Один із поширених способів – використовувати префікс у URI для позначення версії API. Наприклад, API версії 1.0 може мати URI /v1/users, а API версії 2.0 може мати URI /v2/users.

Інший спосіб версіонування RESTful API – використовувати заголовок HTTP Accept. Клієнт може зазначити в цьому заголовку версію API, яку він хоче

використовувати. Наприклад, клієнт може відправити заголовок Accept: application/vnd.myapi.v1+json, щоб зазначити, що він хоче використовувати API версії 1.0.

Третій спосіб версіонування RESTful API – використовувати заголовок HTTP X-API-Version. Сервер може зазначити в цьому заголовку версію API, яку він використовує. Клієнт може перевірити цей заголовок, щоб переконатися, що він використовує правильну версію API.

Який спосіб версіонування RESTful API вибрати, залежить від конкретних потреб. Якщо існує необхідність, щоб клієнти могли легко визначити, яку версію API вони використовують, можна використовувати префікс у URI. Якщо необхідно дати клієнтам більше контролю над тим, яку версію API використовувати, можна використовувати заголовок HTTP Accept. Якщо необхідно, щоб сервер міг повідомляти клієнтам, яка версія API використовується, можна використовувати заголовок HTTP X-API-Version.

Захист RESTful API є важливим для запобігання несанкціонованому доступу, зміни або видалення даних. Є кілька способів захистити RESTful API, враховуючи:

- *Аутентифікацію та авторизацію*. Аутентифікація перевіряє, чи є користувач дійсним, а авторизація визначає, чи має користувач дозволи на доступ до ресурсу. Для RESTful API найчастіше використовуються аутентифікація на основі токена і авторизація на основі ролей.
- *Шифрування* даних захищає їх від несанкціонованого доступу. Для RESTful API часто використовується TLS (SSL) для шифрування даних, що передаються між клієнтом і сервером.
- *Контроль доступу* визначає, хто може отримати доступ до ресурсу. Для RESTful API можна використовувати політику ACL (Access Control List) для обмеження доступу до ресурсів.

- *Відстеження* може допомогти у виявленні атак на RESTful API. Для RESTful API можна використовувати журнали для відстеження доступу до ресурсів.

Додаткові поради щодо захисту RESTful API: використання міцних паролей і токенів; підтримання в актуальному стані програмного забезпечення; виконання сканування безпеки для виявлення вразливостей у RESTful API.

Приклади того, як можна захистити RESTful API:

- Для аутентифікації користувачів можна використовувати токени, що генеруються сервером. Токени можна зберігати в cookie-файлах або в локальному сховищі браузера.
- Для шифрування даних, що передаються між клієнтом і сервером, можна використовувати TLS (SSL). TLS шифрує дані за допомогою алгоритму симетричного шифрування, такого як AES, а потім передає їх за допомогою алгоритму асиметричного шифрування, такого як RSA.
- Для контролю доступу до ресурсів можна використовувати політику ACL (Access Control List). ACL визначає, які користувачі мають доступ до певного ресурсу.
- Для відстеження доступу до ресурсів можна використовувати журнали. Журнали можна використовувати для виявлення підозрілої активності, наприклад, надмірної кількості запитів від однієї IP-адреси.

Визначення різних методів аутентифікації

Аутентифікація – це процес підтвердження особи користувача. RESTful API можуть використовувати різні методи аутентифікації, щоб гарантувати, що лише авторизовані користувачі мають доступ до ресурсів API.

Найпоширеніші методи аутентифікації RESTful API:

- *Базова аутентифікація* – це найпростіший метод аутентифікації. Він використовує логін і пароль користувача, які передаються в HTTP-заголовку Authorization. Базова аутентифікація є не дуже безпечним методом аутентифікації,

оскільки логін і пароль користувача пересилаються в незашифрованому вигляді.

Приклад HTTP-запиту GET із використанням базової аутентифікації:

GET /users/1 HTTP/1.1

Authorization: Basic dXNlcjpwYXNz

Цей запит використовує логін і пароль "user:password". Логін і пароль передаються в HTTP-заголовку Authorization у вигляді базової аутентифікації.

- *OAuth* – це протокол авторизації, який дозволяє користувачам авторизувати доступ до своїх ресурсів третім сторонам без розкриття своїх облікових даних. OAuth є більш безпечним методом аутентифікації, ніж базова аутентифікація, оскільки пароль користувача не передається в незашифрованому вигляді. Для OAuth клієнт повинен отримати доступний токен OAuth від сервера. Цей токен можна отримати, авторизувавши користувача на сервері.

Наприклад, клієнт може використовувати HTTP-запит POST до URI /oauth/authorize для авторизації користувача. Цей запит повинен містити HTTP-заголовок Authorization, який має значення Basic <username>:<password>.

Якщо авторизація успішна, сервер поверне клієнту токен OAuth. Клієнт може потім використовувати цей токен для доступу до ресурсів API.

- *JWT* – це формат обміну інформацією, який дозволяє обмінюватися зашифрованими даними між двома сторонами. JWT часто використовується для аутентифікації RESTful API. JWT є більш безпечним методом аутентифікації, ніж базова аутентифікація або OAuth, оскільки дані, які передаються, зашифровані. Для JWT клієнт повинен одержати JWT від сервера. Цей JWT можна одержати, авторизувавши користувача на сервері.

Наприклад, клієнт може використовувати HTTP-запит POST до URI /jwt/authenticate для авторизації користувача. Цей

запит повинен містити HTTP-заголовок Authorization, який має значення Basic <username>:<password>.

Якщо авторизація успішна, сервер поверне клієнту JWT. Клієнт може потім використовувати цей JWT для доступу до ресурсів API.

- *API-ключі* – це унікальні ідентифікатори, які видаються користувачам для доступу до RESTful API. API-ключі є більш безпечним методом аутентифікації, ніж базова аутентифікація, оскільки вони не передаються в незашифрованому вигляді. Для API-ключів клієнт повинен одержати API-ключ від сервера. Цей API-ключ можна одержати, зареєструвавшись на сервері.

Наприклад, клієнт може використовувати HTTP-запит POST до URI /api-keys/create для одержання API-ключа.

Якщо реєстрація успішна, сервер поверне клієнту API-ключ. Клієнт може потім використовувати цей API-ключ для доступу до ресурсів API.

- *Глобальна аутентифікація* – це метод аутентифікації, який використовує зовнішню систему аутентифікації, наприклад, LDAP або Active Directory. Глобальна аутентифікація є більш зручним методом аутентифікації для користувачів, оскільки їм не потрібно запам’ятовувати додаткові облікові дані. Для глобальної аутентифікації клієнт повинен бути аутентифікований на зовнішній системі аутентифікації, наприклад, LDAP або Active Directory.

Наприклад, клієнт може використовувати HTTP-запит POST до URI /ldap/authenticate для аутентифікації на сервері LDAP.

Якщо аутентифікація успішна, сервер видасть клієнту токен, який можна використовувати для доступу до ресурсів API.

Використання механізмів авторизації для забезпечення доступу до ресурсів

Авторизація – це процес надання користувачеві дозволів на доступ до певних ресурсів. RESTful API можуть використовувати різні механізми авторизації для забезпечення доступу до ресурсів.

Найпоширеніші механізми авторизації RESTful API:

- *Рольова авторизація* базується на ролі, яку користувач виконує в системі. Кожен користувач має одну або кілька ролей, і кожна роль надає певні дозволи. Наприклад, користувач із роллю «адміністратор» може мати доступ до всіх ресурсів API, а користувач із роллю «користувач» може мати доступ лише до певних ресурсів.

Для реалізації рольової авторизації в RESTful API можна використовувати HTTP-заголовок Authorization. Цей заголовок можна використовувати для передавання токена або іншого маркера, який є користувачем.

Наприклад, для того щоб одержати доступ до ресурсу /users/johndoe, користувач із роллю «адміністратор» може відправити такий HTTP-запит:

GET /users/johndoe HTTP/1.1

Authorization: Bearer admin

Цей запит буде успішним, якщо користувач має роль «адміністратор».

- *Політика ACL (Access Control List)* – це список дозволів, які надаються для конкретного ресурсу. Ці дозволи можуть визначатися в коді API або в зовнішній системі управління доступом. Наприклад, для ресурсу «користувач» може бути визначено, що користувачі з роллю «адміністратор» можуть читати, оновлювати та видаляти користувачів, а користувачі з роллю «користувач» можуть лише читати користувачів.

Для реалізації політики ACL в RESTful API можна використовувати HTTP-заголовок Access-Control-Allow-Origin.

Цей заголовок можна використовувати для позначення, з яких доменів дозволено доступ до ресурсу.

Наприклад, для того щоб дозволити доступ до ресурсу /users/johndoe лише з домену example.com, можна використовувати такий HTTP-заголовок Access-Control-Allow-Origin:

Access-Control-Allow-Origin: example.com

Цей запит буде успішним, якщо запит був відправлений із домену example.com.

- *Атрибутна авторизація.* Атрибутна авторизація базується на атрибутах користувача. Атрибути користувача можуть містити інформацію, таку як логін, ім'я, роль, група або організація. За допомогою атрибутної авторизації можна надавати користувачам різні дозволи на основі їхніх атрибутів. Наприклад, користувачеві з атрибутом «географічне місце розташування США» може бути дозволено доступ до ресурсів, доступних лише в США.

Для реалізації атрибутної авторизації в RESTful API можна використовувати HTTP-заголовки Authorization і Access-Control-Allow-Origin. Ці заголовки можна використовувати для передавання атрибутів користувача і дозволів, наданих для ресурсу.

Наприклад, для того щоб надати користувачеві з атрибутом «географічне місце розташування США» доступ до ресурсу /users/johndoe, можна використовувати такі HTTP-заголовки:

Authorization: Bearer {token}
Access-Control-Allow-Origin: example.com
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: Authorization, Geolocation

Цей запит буде успішним, якщо користувач має атрибут «географічне місце розташування США» і запит був відправлений із домену example.com.

- *Багатофакторна аутентифікація (MFA)* потребує від користувача подати два або більше факторів ідентифікації

для аутентифікації. Ці фактори можуть містити пароль, одноразовий код, відбиток пальця або розпізнавання обличчя. MFA може допомогти захистити API від несанкціонованого доступу.

Вибір механізму авторизації RESTful API залежить від конкретних потреб API. Важливо вибрати механізм авторизації, який буде ефективним і безпечним для захисту ресурсів API.

MFA можна реалізувати в RESTful API за допомогою різних методів. Один із найпоширеніших методів – це використання HTTP-заголовка Authorization. Цей заголовок можна використовувати для передавання фактору ідентифікації, такого як одноразовий код, у запиті.

Наприклад, для того щоб підтвердити авторизацію користувача за допомогою одноразового коду, отриманого за допомогою SMS, можна використовувати такий HTTP-заголовок Authorization:

Authorization: Bearer {token}

Цей запит буде успішним, якщо одноразовий код, переданий у заголовку Authorization, збігається з одноразовим кодом, одержаним користувачем за допомогою SMS.

Журналювання діяльності API

Журналювання діяльності API – це процес запису всіх запитів і відповідей, які надходять до API. Це корисно для відстеження використання API, виявлення проблем і забезпечення відповідності вимогам безпеки.

Існує декілька різних способів журналювання діяльності API. Один із найпоширеніших способів – це використання системи логування сервера. Системи логування сервера зазвичай зберігають журнали в файлах або базі даних. Інший спосіб журналювання діяльності API – це використання спеціалізованої системи журналювання API. Ці системи зазвичай пропонують більше функцій, ніж системи логування сервера, такі як аналіз та оброблення журналів. Приклади таких ситсем: Apache Log4j або ELK Stack. Інший спосіб – використовувати API-шлюз, який може журналювати запити та відповіді автоматично.

Журналювання діяльності API – це процес збереження інформації про всі запити та відповіді API, які надходять до системи. Ця інформація може містити такі дані, як час запиту, хост, IP-адреса, користувач, метод, шлях, запитані параметри та відповіді.

Журналювання діяльності API має низку переваг, зокрема:

- покращання безпеки. Журнали діяльності API можна використовувати для моніторингу та виявлення несанкціонованих доступів до системи»
- покращання продуктивності. Журнали діяльності API можна використовувати для виявлення проблем із продуктивністю та оптимізації системи;
- покращання обслуговування клієнтів. Журнали діяльності API можна використовувати для розслідування проблем клієнтів і надання їм підтримки.

Приклади того, як можна використовувати журналювання діяльності API

- Компанія, яка надає послуги хмарного зберігання, може використовувати журналювання діяльності API для моніторингу доступу до своїх серверів. Якщо журнали діяльності показують, що хтось намагається одержати несанкціонований доступ до сервера, компанія може вжити заходів для захисту системи.
- Компанія, яка розробляє програмне забезпечення, може використовувати журналювання діяльності API для виявлення проблем із продуктивністю свого програмного забезпечення. Якщо журнали діяльності показують, що запити до певного API займають занадто багато часу, компанія може вжити заходів для оптимізації API.
- Компанія, яка надає клієнтську підтримку, може використовувати журналювання діяльності API для розслідування проблем клієнтів. Якщо клієнт повідомляє про проблему з API, компанія може проаналізувати журнали діяльності, щоб визначити причину проблеми.

Вимірювання продуктивності API для виявлення проблем та оптимізації

Вимірювання продуктивності API є важливим аспектом управління API. Це допомагає власникам API зрозуміти, як їхні API працюють, і виявити потенційні проблеми. Існує кілька різних показників продуктивності API, які можна виміряти. Деякі з найпоширеніших показників мають:

- час відповіді – це час, який потрібно API для оброблення запиту та надання відповіді;
- частота запитів – це кількість запитів, які надходять до API за певний період часу;
- ширина смуги пропускання – це кількість даних, які обробляються API за певний період часу;
- стосовність помилок – це кількість запитів, які завершилися помилкою;
- вимірювання цих показників можна зробити за допомогою різних інструментів і методів. Деякі поширені інструменти містять:
  - *аналізатори продуктивності API*. Ці інструменти можуть моніторити API та збирати дані про продуктивність;
  - *системи логування API*. Ці системи можуть зберігати журнали запитів і відповідей API, які можна використовувати для аналізу продуктивності.

Приклади того, як можна використовувати вимірювання продуктивності API для виявлення проблем та оптимізації:

- Компанія, яка надає послуги хмарного зберігання, може використовувати вимірювання продуктивності API для виявлення проблем із продуктивністю своїх серверів. Якщо час відповіді API значно збільшується, компанія може вжити заходів для оптимізації серверів.
- Компанія, яка розробляє програмне забезпечення, може використовувати вимірювання продуктивності API для виявлення проблем із продуктивністю свого програмного забезпечення. Якщо частота запитів до певного API значно збільшується, компанія може вжити заходів для оптимізації API.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Мартін Р. Чиста архітектура: мистецтво розроблення програмного забезпечення / Р. Мартін ; пер. з англ. І. Бондар-Терещенко. – Харків : Ранок; Фабула, 2019. – 368 с.

2. Bridging the Gap between Requirements Engineering and Software Architecture [Електронний ресурс] : A Problem-Oriented and Quality-Driven Method / A. Alebrahim ; by Azadeh Alebrahim. – 1st ed. 2017. – Wiesbaden : Springer Fachmedien Wiesbaden, 2017. – XXVI, 500 p. 141 illus.

3. Web Service Implementation and Composition Techniques [Електронний ресурс] / H. Paik, A. L. Lemos, M. C. Barukh etc. ; by Hye-young Paik, Angel Lagares Lemos, Moshe Chai Barukh, Boualem Benatallah, Aarthi Natarajan. – 1st ed. 2017. – Cham : Springer International Publishing, 2017. – XIII, 256 p. 102 illus., 80 illus. in color.

4. Сучасні інформаційні технології і системи [Електронний ресурс] : монографія / В. П. Бурдаєв, Н. Г. Аксак, М. В. Кушнарьов та ін. ; за заг. ред . В. С. Пономаренка. – Харків : Технічний коледж Луцького НТУ, 2021. – 182 с.

5. Muller G. Systems Architecting [Електронний ресурс]. – 2023. – 268 с. Режим доступу : https://www.gaudisite.nl/SystemArchitectureBook.pdf (дата доступу: 30.06.2023).

6. Software Design and Architecture Specialization [Електронний ресурс]. – Режим доступу : https://www.coursera.org/specializations/software-design-architecture#courses (дата доступу: 30.06.2023).

7. API Security Tutorial [Електронний ресурс] . – Режим доступу : https://www.wallarm.com/what/api-security-tutorial (дата доступу: 30.06.2023).

Електронне навчальне видання

Бойко Ольга Василівна

ІНТЕГРОВАНІ ІНФОРМАЦІЙНІ СИСТЕМИ

Конспект лекцій
для студентів спеціальності *122 «Комп'ютерні науки»*
освітнього ступеня «магістр»
усіх форм навчання

Відповідальний за випуск С. М. Ващенко
Редактор Н. М. Мажуга
Комп'ютерне верстання О. В. Бойко

Формат 60х84/16. Ум. друк. арк. 7,68. Обл.-вид. арк. 7,25.

Видавець і виготовлювач
Сумський державний університет,
вул. Римського-Корсакова, 2, м. Суми, 40007
Свідоцтво суб'єкта видавничої справи ДК № 3062 від 17.12.2007.