6. Архітектура веб-додатків

Архітектура програмного забезпечення - структура, на базі якої створюється додаток, взаємодіють його модулі та компоненти.

Архітектура веб-додатку є макетом з усіма програмними компонентами (такими як бази даних, додатки та проміжне ПЗ) та їх взаємодією один з одним. Архітектура визначає, як дані доставляються через HTTP, і гарантує, що сервер на стороні клієнта і внутрішній сервер можуть їх зрозуміти. Забезпечує наявність достовірних даних у всіх запитах користувачів, створення записів та керування ними, доступ та аутентифікацію на основі дозволів.

Архітектурний шаблон — це спосіб вирішення певного завдання з проектування програмного забезпечення. Вони використовуються при простому написанні коду, створенні класів та плануванні їхньої взаємодії. Архітектурні шаблони задіяні на вищому рівні абстракції - при плануванні взаємодії користувача програми із сервером, даними та іншими компонентами проекту.

Компоненти архітектури веб-додатків

Зазвичай архітектура веб-додатків містить 3 основні компоненти:

- **Веб-браузер.** Клієнтський компонент або інтерфейсний компонент є ключовим компонентом, який взаємодіє з користувачем, отримує вхідні дані та керує логікою подання, контролюючи взаємодію користувача з додатком. Вхідні дані користувача також перевіряються, якщо це необхідно.
- **Веб-сервер.** Внутрішній компонент або компонент на стороні сервера, обробляє бізнес-логіку та обробляє запити користувачів, надсилаючи запити до потрібного компонента та керуючи всіма операціями програми. Він може запускати і контролювати запити від різних клієнтів.
- **Сервер бази даних.** Надає необхідні дані для програми. Він опрацьовує завдання, пов'язані з даними. У багаторівневій архітектурі сервери баз даних можуть керувати бізнес-логікою за допомогою збережених процедур.

Трирівнева архітектура

У традиційній дворівневій архітектурі є два компоненти, а саме клієнтська система або інтерфейс користувача і внутрішня система, яка зазвичай представляє сервер бази даних. Тут бізнес-логіка вбудована в інтерфейс користувача або сервер бази даних. Недоліком дворівневої архітектури і те, що зі збільшенням кількості користувачів зменшується продуктивність. Крім того, пряма взаємодія бази даних і пристрою користувача також викликає деякі проблеми з безпекою.

Більш надійною в плані захисту інформації є трирівнева архітектури, є представлено три рівні.

Рис. Трирівнева архітектура веб-додатку

1. Рівень подання / Клієнтський рівень
2. Рівень логіки / Бізнес-рівень
3. Рівень даних

У цій моделі проміжні сервери одержують запити клієнтів та обробляють їх, координуючи свої дії з підлеглими серверами, застосовуючи бізнес-логіку. Зв'язком між клієнтом та базою даних управляє проміжний рівень логіки, що дозволяє клієнтам отримувати доступ до даних різних рішень СУБД. Трирівнева архітектура безпечніша, оскільки клієнт не має прямого доступу до даних. Можливість розгортання серверів додатків на кількох комп'ютерах забезпечує більш високу масштабованість, кращу продуктивність та краще повторне використання.

Цілісність даних покращується, оскільки всі дані проходять через сервер додатку, який вирішує, як і ким слід здійснювати доступ до даних.

Рівні сучасної архітектури веб-додатків

Створення багаторівневої архітектури сучасної веб-програми допомагає визначити роль кожного компонента програми та легко вносити зміни у відповідний рівень, не торкаючись додатку в цілому. Це дозволяє легко писати, налагоджувати, керувати та повторно використовувати код.

Рівень подання: клієнтський компонент (зовнішній інтерфейс)

Архітектура веб-додатку дозволяє користувачам взаємодіяти з сервером та серверною службою через браузер. Код знаходиться у браузері, отримує запити та надає користувачеві необхідну інформацію. Саме тут користувач бачить дизайн UI/UX, інформаційні панелі, повідомлення, конфігураційні налаштування, макет та інтерактивні елементи. Ось деякі з інтерфейсних технологій, що найчастіше використовуються:

- **HTML.** Стандартна мова розмітки, яка дозволяє розробникам структурувати вміст веб-сторінки за допомогою низки елементів сторінки.
- **CSS.** Каскадні таблиці стилів - це популярна мова таблиць стилів, яка дозволяє розробникам розділяти вміст та макет веб-сайтів для сайтів, розроблених за допомогою мов розмітки. Використовуючи CSS, можна визначити стиль елементів і повторно використовувати їх кілька разів.
- **JavaScript.** Популярна мова програмування на стороні клієнта, яка останнім часом використовується більш ніж на 90% веб-сайтів. Всі браузери містять движок JS для

запуску коду JavaScript на пристроях. Використовуючи JS на клієнті зменшує навантаження на сервер.

- **React.** JS-фреймворк з відкритим кодом, що набрав популярність в останні роки. React дозволяє розробникам легко створювати високоякісні динамічні веб-додатки з мінімальними витратами коду та зусиль. Він простий у освоєнні та використанні. Для розробників доступна велика документація та багато зручних інструментів. Код можна використати повторно.
- **Vue.** JS-фреймворк, що дозволяє розробникам легко створювати інтерфейси інтерфейсу користувача для Інтернету, настільних комп'ютерів і мобільних пристроїв. Vue.js поставляється із зручними інструментами, які задовольняють базові потреби програмування, легко інтегрується з наявними програмами. Документація лаконічна та добре структурована.
- **Angular.** JS-фреймворк для веб-додатків з відкритим вихідним кодом, один з популярних середовищ розробки інтерфейсів, доступних на ринку. Пропонує всі функції для розробки додатків, таких як компоненти, модулі, шаблони, директиви, впровадження сервісів та залежностей, маршрутизація тощо. Допомагає розробникам швидко створювати прототипи та використовує прості шаблони HTML. Тестування виконується швидко та легко завдяки стилю архітектури застосування залежностей. Поставляється з різними плагінами, інструментами із коробки.

Рівень подання: серверний компонент (внутрішня частина)

Серверний компонент – це ключовий компонент архітектури веб-додатку, який отримує запити користувачів, виконує бізнес-логіку та доставляє необхідні дані з зовнішніх систем. Він містить сервери, бази даних, веб-сервіси тощо. Ось деякі з серверних технологій, що найчастіше використовуються:

- **Java.** Популярна і ефективна об'єктно-орієнтована мова програмування на основі класів, яка дозволяє розробникам писати код і запускати його на будь-якій платформі, використовуючи середовище віртуальної машини Java. Мова проста у вивченні, програмуванні, компіляції та налагодженні.
- **Python.** Мова програмування високого рівня з відкритим вихідним кодом, проста у освоєнні та розробці і має багато функцій. Ця мова підходить для невеликих і великих проектів веб-додатків, а також для різних сегментів, таких як мобільні додатки, відеоігри, програмування штучного інтелекту тощо. Python пропонує велику бібліотеку, яка містить код майже для всіх типів програм. Python використовують популярні IT-гіганти, такі як Google, Spotify, Instagram і Facebook тощо.
- **Ruby.** Популярна мова програмування, у поєднанні з інфраструктурою Rails дозволяє розробникам швидко створювати та розгортати програми. Інструмент пропонує велику бібліотеку та корисні інструменти. Ruby постачається з вбудованою системою безпеки для зменшення ризиків, пов'язаних із SQL-ін'єкціями, програмним забезпеченням для міжсайтових сценаріїв (XSS) та підробкою міжсайтових запитів (CSRF).
- **Golang.** Мова програмування Go також відома як Golang, схожа на мову C, проста у вивченні та використанні. Оскільки немає віртуальних середовищ виконання, код Go компілюється швидше і створює менший двійковий файл. Стандартна бібліотека пропонує низку вбудованих функцій, а також підтримку тестування.

- **Laravel.** Це PHP-фреймворк з відкритим вихідним кодом. Завдяки широким вбудованим функціям та структурам розробники можуть легко писати код і швидше розгортати програми, що підвищує продуктивність та швидкість. Важливою перевагою PHP Laravel є його функція автоматичного тестування, яка допомагає тестувати та усувати помилки на початковому етапі.
- **Node.js.** Це кросплатформне середовище виконання з відкритим вихідним кодом, оскільки пропонує багату бібліотеку модулів JavaScript, які дозволяють розробникам швидко створювати якісні програми. Node.js не буферизує дані та виконує код дуже швидко. Він керується подіями, асинхронний і працює в одному потоці, володіючи при цьому високою масштабованістю. Node.js найкраще підходить для програм з потоковою передачею даних, інтенсивного використання даних, пов'язаних із введенням-виводом та заснованих на JSON-API.
- **.NET.** Середовище розробки програмного забезпечення для настільних та веб-додатків. .NET представляє модель об'єктно-орієнтованого програмування і використовує модульну структуру, яка дозволяє розробникам розділяти код на дрібні частини, безперешкодно створювати програмні продукти та керувати ними за допомогою конвеєрів CI/CD. Пропонує надійну і водночас просту систему кешування, яка підвищує швидкість та продуктивність.

Рівень подання: інтерфейс прикладного програмування (API)

Інтерфейс прикладного програмування (API) - це не технологія, а концепція, що дозволяє розробникам отримувати доступ до певних даних та функцій програмного забезпечення. Це посередник, який дозволяє програмам спілкуватися один з одним. Він містить протоколи, інструменти та визначення підпрограм, необхідні для створення додатків.

Наприклад, коли користувач входить до додатку, він викликає API для отримання даних та облікових даних свого облікового запису. Додаток зв'яжеться з відповідними серверами, щоб отримати цю інформацію і повернути ці дані користувача додатку.

Веб-API – це API, доступний через Інтернет за протоколом HTTP. Його можна побудувати з використанням таких технологій, як .NET, Java, PHP тощо. Завдяки розробникам API не потрібно створювати все з нуля, а використовувати існуючі функції, представлені у вигляді API, для підвищення продуктивності та прискорення виходу на ринок. Скорочуючи зусилля розробки, API значно скорочують витрати на розробку. Це також покращує спільну роботу та зв'язок в екосистемі, підвищуючи якість обслуговування клієнтів.

Існують різні типи API

- RESTful API: API передачі репрезентативного стану у полегшеному форматі JSON. Він добре масштабується, надійний та забезпечує високу продуктивність, що робить його найпопулярнішим API.
- SOAP: простий протокол доступу до об'єктів використовує XML для передачі даних. Це вимагає більшої пропускної спроможності та розширеної безпеки
- XML-RPC: мова розмітки, що розширюється — віддалені виклики процедур використовують спеціальний формат XML для передачі даних.
- JSON-RPC: використовує формат JSON для передачі даних.

Рівень логіки: екземпляр сервера/екземпляр хмари

Сервери або хмарні екземпляри є важливою частиною архітектури веб-додатків. Хмарний екземпляр — це екземпляр віртуального сервера, який створюється та розміщується у загальнодоступній або приватній хмарі та доступний через Інтернет. Він працює як фізичний сервер, який може легко переміщатися між кількома пристроями або розгортати кілька екземплярів на одному сервері. Таким чином, він відрізняється високою динамічністю, масштабованістю та економічністю. Можна автоматично замінювати сервери без простою програм. Використовуючи хмарні екземпляри, можна легко розгортати веб-додатки та керувати ними у будь-якому середовищі.

Рівень даних: база даних

База даних — це ключовий компонент веб-програми, яка зберігає інформацію для веб-додатку та керує нею. Використовуючи базу даних, можна шукати, фільтрувати та сортувати інформацію на основі запиту користувача та надавати необхідну інформацію кінцевому користувачеві. База даних забезпечує доступ з врахуванням ролей, підтримує цілісність даних. При виборі бази даних для архітектури веб-додатку потрібно уважно вважати на 4 аспекти: розмір, швидкість, масштабованість та структура.

Для структурованих даних добрим вибором є бази даних на основі SQL. Вона підходить для додатків, в яких цілісність даних є ключовою вимогою.

Для обробки неструктурованих даних добрим варіантом є NoSQL. Вона підходить для додатків, у яких характер вхідних даних непередбачуваний.

Типи архітектур веб-додатків

Архітектуру веб-додатків можна розділити на різні категорії в залежності від шаблонів розробки та розгортання програмного забезпечення.

Побудова архітектури веб-додатку - це комплекс заходів, спрямованих на те, щоб чітко визначити, як буде побудована система. Важливо розуміти, що немає ідеального рішення при побудові архітектури, все залежить від конкретного проекту, його завдань і від того, як працює веб-додаток.

Для визначення архітектури проекту потрібно:

- Технічне завдання.
- Архітектор або людина з багатим досвідом розробки.
- Розуміння дедлайнів.
- Розуміння життєвого циклу проекту.
- Розуміння навантаження.

Щоб визначитися з повною архітектурою проекту, потрібно проаналізувати багато інформації. Але для визначення типу архітектури іноді вистачає лише ідеї проекту.

Загалом відокремлюють три види архітектури веб-додатків:

- Монолітна.
- Мікросервісна.
- Безсерверна.

Монолітна архітектура

Монолітна архітектура — це традиційна модель розробки програмного забезпечення, також відома як архітектура веб-розробки, де все програмне забезпечення розробляється як єдиний фрагмент коду, що проходить через традиційну водоспадну модель. Всі компоненти архітектури взаємозалежні та взаємопов'язані, і кожен компонент необхідний для запуску програми. Щоб змінити або оновити конкретну функцію, потрібно змінити весь код, який буде переписано та скомпільовано.

Оскільки монолітна архітектура розглядає весь код як єдину програму, створення нового проекту, застосування фреймворків, скриптів та шаблонів, а також тестування стають простими та легкими. Розгортання також просте. Однак у міру того, як код стає більшим, їм стає важко керувати або вносити оновлення; навіть для невеликої зміни потрібно пройти весь процес архітектури веб-розробки.

Оскільки кожен елемент є взаємозалежним, масштабувати програму непросто. Більше того, це ненадійно, оскільки єдина точка відмови може вивести програму з ладу. Монолітна архітектура буде доречною, якщо потрібно створити легкий за вагою додаток з обмеженим бюджетом, а команда розробників працює з одного місця, а не віддалено.

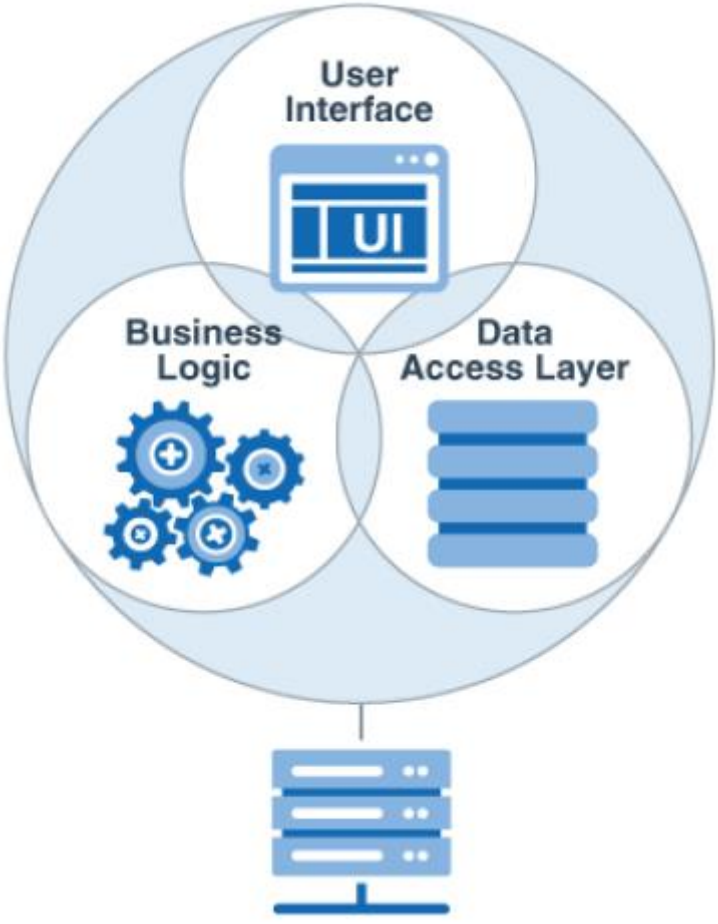

Рис. Монолітна архітектура веб-додатку

Загалом, вважається, що це застаріле архітектурне рішення, але таке рішення має свої переваги, і воно дуже добре підходить при побудові деяких проектів.

Переваги:

- Простота розгортання. Монолітну архітектуру можна швидко і відносно просто розгортати, оскільки зазвичай є єдина точка входу.
- Розробка. Розробка монолітної архітектури зазвичай відбувається швидко, оскільки всі компоненти та модулі знаходяться в одній кодовій базі і завжди під рукою.
- Налагодження. Налагодження монолітної архітектури спрощено за рахунок того, що все поряд, і є можливість відстежити всі ланки виконання коду.

Недоліки:

- Масштабування. Монолітні архітектури масштабуються лише повністю, тобто якщо навантаження зростає на один модуль, не можна масштабувати тільки цей модуль, потрібно масштабувати весь моноліт.
- Надійність. Якщо моноліт виходить з ладу, то виходить весь цілком.
- Зміна та оновлення технологій. У моноліті це майже неможливо.
- Недостатня еластичність. Монолітні архітектури негнучкі, тобто зміна одного модуля в моноліті майже завжди впливатиме на інший модуль.

Мікросервісна архітектура

Архітектура мікросервісів вирішує кілька проблем, що виникають у монолітній архітектурі. Це архітектурне рішення, яке базується на розподілі модулів на окремі системи, які спілкуються між собою за допомогою повідомлень. Вся система - це набір маленьких систем, пов'язаних між собою, детальніше проілюстровано на рис.

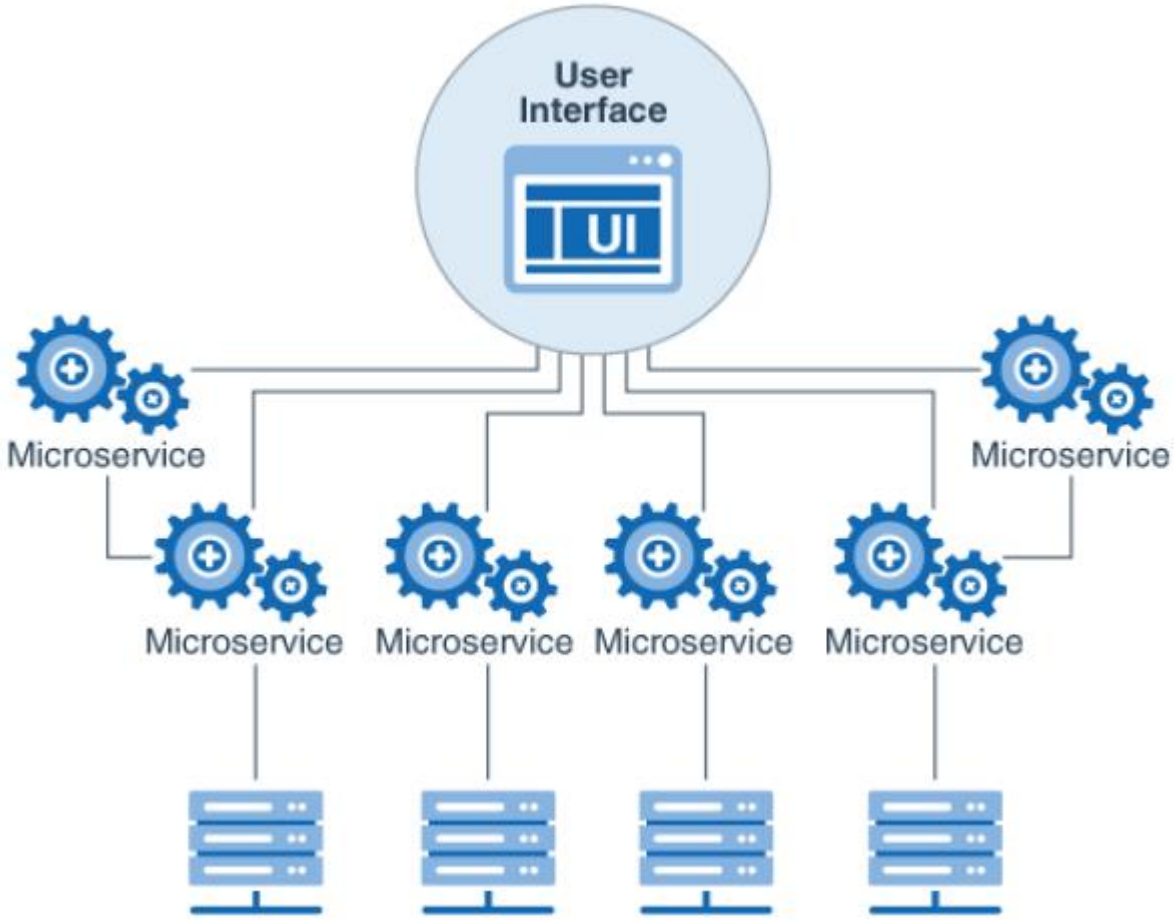

Рис. Мікросервісна архітектура веб-додатку

Кожен мікросервіс містить власну базу даних та використовує певну бізнес-логіку, що означає, що можна легко розробляти та розгортати незалежні сервіси.

Оскільки архітектура мікросервісів слабко пов'язана, вона забезпечує гнучкість для оновлення/модифікації та масштабування незалежних сервісів. Розробка стає простою та ефективною, а безперервні доповнення стають можливими. Розробники можуть швидко адаптуватись до інновацій.

Мікросервісна архітектура — доречний вибір для високомасштабованих та складних програм. Проте, розгортання кількох служб з екземплярами середовища виконання є складним завданням. Зі зростанням кількості сервісів зростає і складність керування ними. Крім того, програми мікросервісів спільно використовують бази даних розділів. Це означає, що потрібно забезпечити узгодженість між кількома базами даних, на які впливає транзакція.

Це дуже модна і популярна архітектура, але має свої недоліки, не підходить для маленьких і середніх проектів.

Переваги:

- Гнучкість. Мікросервісна архітектура дуже гнучка завдяки тому, що кожен сервіс є самостійною системою, тому зміни в ній можуть вплинути тільки на неї.
- Масштабування. На відміну від монолітної архітектури, можна масштабувати лише певну частину системи, оскільки вона є самостійною.
- Гнучкість технологій. У мікросервісній архітектурі кожна з підсистем може бути реалізована будь-якою мовою програмування та за допомогою будь-яких технологій.
- Надійність. Звичайний вихід з ладу однієї з підсистем не ламає всю систему загалом.

Недоліки:

- Процес розробки. Мікросервісну архітектуру непросто розробляти, оскільки потрібно виробляти кілька підсистем і налагодити взаємодію між ними.
- Комунікація команд. Часто буває так, що одна команда розробляє одну підсистему, тому потрібно мати комунікацію між командами, щоб налагодити взаємодію між підсистемами.
- Налагодження. Мікросервісну архітектуру складно налагоджувати, оскільки потрібно знайти, який сервіс зламався і чому.
- Розгортання. Початкове розгортання є непростим, і додавання нових сервісів вимагає налаштування ключових частин проекту.

Безсерверна архітектура (Serverless)

Безсерверна архітектура – це альтернатива мікросервісам, яка автоматизує все розгортання завдяки хмарним технологіям. Докладніше проілюстровано на зображенні. З назви можна подумати, що тут відсутня серверна частина, але це не так, тут відсутня робота із сервером як із середовищем.

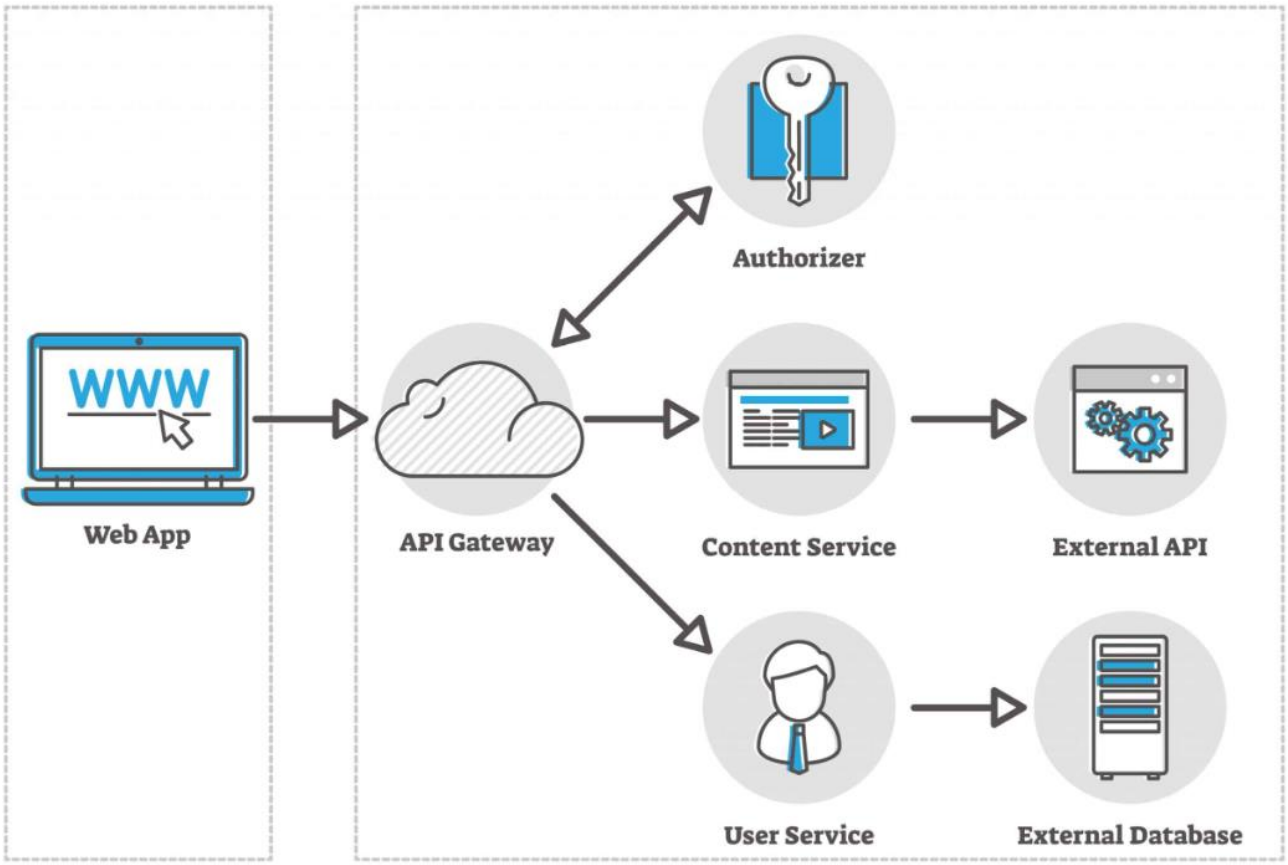

Рис. Безсерверна архітектура веб-додатку

Безсерверна архітектура – це модель розробки програмних додатків, яка фокусується на розробці замість розгортання та взаємодії між сервісами. У цій структурі забезпечення базової інфраструктури керується постачальником хмарних послуг. Це означає, відбувається оплата тільки за інфраструктуру, коли вона використовується, а не за час простою центрального процесору або простір, що не використовується.

Безсерверні обчислення зменшують витрати, оскільки ресурси використовуються лише під час виконання програми. Завдання масштабування виконуються хмарним провайдером. Бекенд-код спрощується, що зменшує зусилля розробки, витрати і прискорює виведення результатів обробки. Обробка мультимедіа, потокова передача в реальному часі, чат-боти CI Pipelines, повідомлення давачів IoT тощо - ось деякі з варіантів використання безсерверних обчислень.

Переваги:

- Гнучкість. Безсерверна архітектура гнучка за рахунок відокремленості одного модуля від іншого.
- Абстракція від операційної системи. Хмара сама вирішує, яка ОС потрібна та як її потрібно налаштувати.
- Легкий поріг входу. Зазвичай серверлес це дуже простий, відокремлений код, тому розібратися з проектом нескладно.
- Надійність. При правильній побудові проекту надійність така сама, як і у мікросервісній архітектурі.

Недоліки:

- Гнучкість. У розробника обмежений вплив на масштабування, все вирішує хмара.

- Налагодження. Як й у мікросервісній архітектурі, налагодження ускладнене взаємодією між компонентами.
- Прив’язка до постачальника послуг (Vendor Lock). Кожна хмара працює за власними правилами, тому переїзд з однієї хмари на іншу майже неможливий.
- Каскадна відмова (Cascade Failur). Це відмова в системі взаємопов’язаних частин, у якій відмова однієї або кількох частин призводить до відмов інших частин, що прогресивно зростає в результаті позитивного зворотного зв’язку. При неправильній побудові проекту компоненти можуть сильно впливати інші компоненти, що призводить до падіння всього проекту.

Висновки

Монолітна архітектура ідеальна для невеликих програм завдяки швидкій розробці, простоті тестування та налагодження, а також низькій вартості. Однак у міру зростання системи вона може стати перешкодою для бізнесу та має еволюціонувати в іншу форму.

Використання **мікросервісів** надає багато переваг. Вони дозволяють створювати більш модульні та зручні у супроводі програми з меншим ризиком поламки при внесенні змін. Надають команді розробників більшої гнучкості у взаємодії над проектом, полегшуючи їм продуктивну роботу на всіх рівнях досвіду — від молодших розробників до тих, хто працює з цими типами систем протягом багатьох років.

Безсерверна архітектура визначає внутрішню структуру програми, тоді як мікросервіси відповідають за побудову програми на макрорівні. Безсерверне застосування може відповідати або не відповідати концепції мікросервісів (хоча це часто рекомендується). Усі або деякі мікросервіси (або жоден) в архітектурі мікросервісів можуть бути створені з використанням технології безсерверної архітектури.

Архітектурні шаблони програмування

На зорі веб-програмування програмісти писали сотні рядків коду безпосередньо в інтерфейсі продукту. Іноді використовувалися сервіси та менеджери для роботи з даними, і тоді рішення виходило з використанням шаблону Document-View. Підтримка такого коду вимагала колосальних витрат, оскільки нового працівника треба було навчити (розповісти), який код за що в продукті відповідає, і про модульне тестування і мови не було. Команда розробки, зазвичай складала 4-5 осіб, які сидять в одній кімнаті.

З часом веб-додатки ставали більшими і складнішими, з однієї згуртованої команди розробників стало багато різних команд розробників, архітекторів, юзабілістів, дизайнерів і програмістів. Тепер кожен відповідальний за свою сферу: графічний інтерфейс (Graphical User Interface), бізнес-логіка, компоненти. З'явився відділ аналізу, тестування, архітектури. Вартість розробки програмного забезпечення зросла в сотні і навіть тисячі разів. Такий підхід до розробки вимагає наявність стійкої архітектури, яка синхронізувала б різні функціональні області продукту між собою.

Шаблони

Сучасні додатки потребують такої різноманітності функцій, що процес їх розробки виріс у розмірі та складності. Щоб зменшити витрати на розробку складного програмного забезпечення необхідно використовувати готові уніфіковані рішення. Шаблонність дій

полегшує комунікацію між розробниками, дозволяє посилатися на відомі конструкції, зменшує кількість помилок.

Шаблон (Design Pattern) - повторна архітектурна конструкція, що представляє собою вирішення проблеми проектування в рамках деякого контексту. Шаблон уточнює та визначає деякі важливі компоненти архітектури програмного забезпечення. Незважаючи на те, що архітектурний шаблон передає образ системи, він не є архітектурою. Фактично, це загальне та багаторазове рішення для типової проблеми в архітектурі програмного забезпечення в певному контексті. Без архітектурного шаблону виникають складнощі під час підтримки бізнес-логіки програми.

Під час розробки програмного забезпечення шаблони архітектурного проектування використовують для вирішення типових проблем.

- Розділити складні завдання на простіші.
- Зменшити помилки.
- Створення коду, який можна тестувати та підтримувати.

Першим головним шаблоном можна вважати – MVC (Model-View-Controller). Це фундаментальний шаблон, який знайшов застосування у багатьох технологіях, надав розвиток новим технологіям і щодня полегшує життя розробникам.

Найбільш поширені види MVC-шаблону, це:

- Model-View-Controller
- Model-View-Presenter
- Model-View- View Model

MVC (Model, View, Controller)

MVC - шаблон (патерн) програмування, що розділяє архітектуру додатка на три модулі і дозволяє змінювати кожен компонент незалежно один від одного.

- **Модель (Model).** Це основна логіка програми. Відповідає за дані, методи роботи з ними та структуру програми. Model реагує на команди з Controller, видає інформацію та/або змінює свій стан. Вона передає дані у View.
- **Подання (View).** Завданням компонента є візуалізація інформації, яку він отримує від моделі. View відображає дані на рівні інтерфейсу користувача. Наприклад, у вигляді таблиці чи списку. View визначає зовнішній вигляд програми та способи взаємодії з ним.
- **Контролер (Controller).** Він забезпечує взаємодію з системою: обробляє дії користувача, перевіряє отриману інформацію та передає її до Model. Controller визначає, як програма буде реагувати на дії користувача. Також Controller може відповідати за фільтрацію даних та авторизацію.

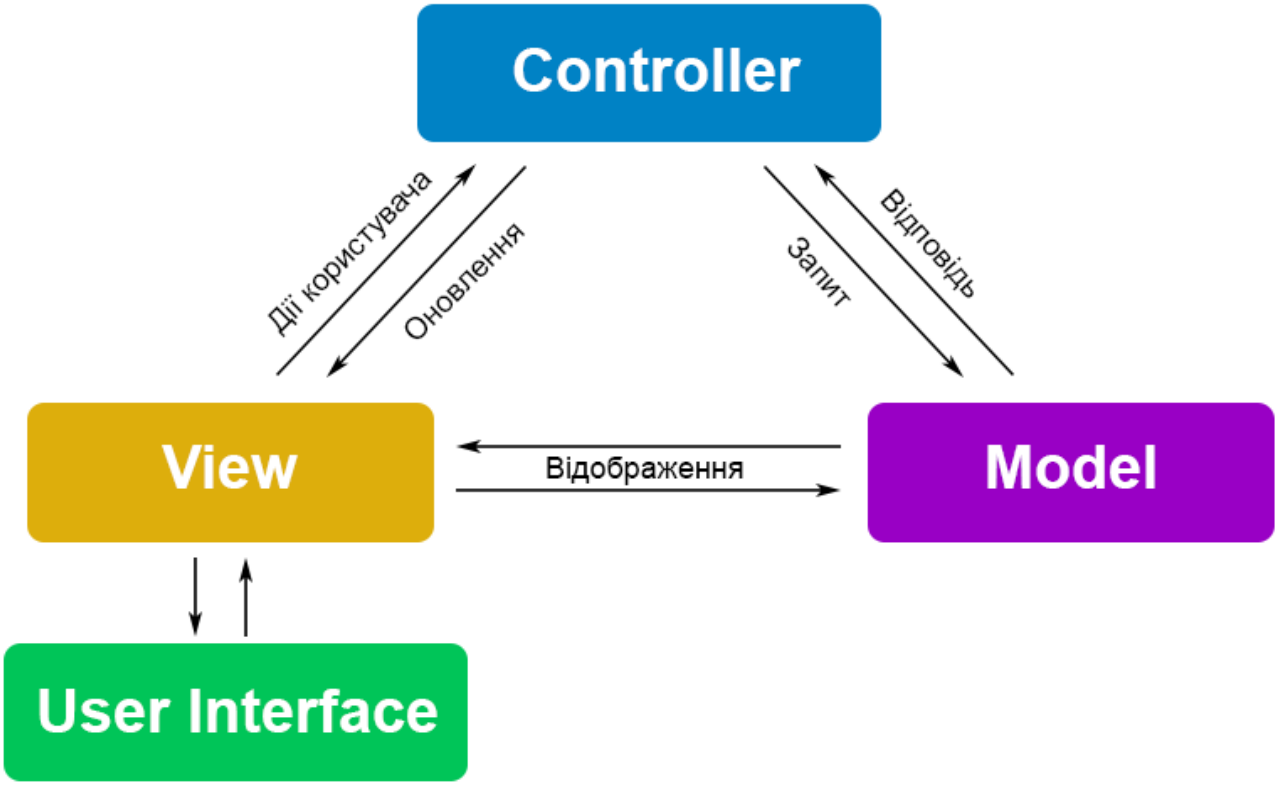

Рис. Логічна схема MVC-шаблону

Компоненти шаблону відрізняються ступенем залежності один від одного та обмеженнями:

- Model не залежить від View та Controller і не може використовувати класи з їхніх розділів.
- View може звертатися до Model за даними та подіями, але не може її змінювати.
- Controller не може відображати дані, але здатний змінювати Model в залежності від дій користувача.

Концепція MVC стала популярна з появою фреймворків, що швидко розгортаються, і інтерактивних веб-додатків.

Розберемо реальному прикладі. Умовна фізична модель MVC-архітектури - персональний комп'ютер, в якому:

- Controller - клавіатура або миша. З їх допомогою користувач вводить команди.
- Model - системний блок, в якому відбувається обробка команд і зберігаються системні та користувацькі файли.
- View - монітор, на якому візуалізується робота системного блоку.

У цьому прикладі легше зрозуміти залежність компонентів друг від друга:

- Якщо замінити системний блок (без перенесення старих даних), але залишити старий монітор та клавіатуру, користувач отримає інший комп'ютер із новою ОС, файлами, системними характеристиками, драйверами тощо.
- Якщо змінити монітор і клавіатуру, але залишити старий системний блок, людина використовуватиме колишню ОС, файли, системні характеристики. Зміни торкнуться лише деяких драйверів.

В архітектурі однієї програми може бути кілька View. Наприклад, в Excel одні й самі дані можна вивести як діаграму чи таблицю. У деяких іграх можна переключити вигляд від першої особи

на вигляд від третьої або взагалі перейти в режим карти. Приклад із веб-розробки — мобільний та десктопний інтерфейс сайтів та програм. Все це різні варіанти View для однієї і тієї ж моделі.

Шаблон MVC дозволяє розділити логічні частини програми та створювати їх незалежно. Тобто писати блоки коду, які можна міняти, не торкаючись інших. Наприклад, переписати спосіб обробки даних, не торкаючись способу їх відображення. Це допомагає ефективно працювати різним програмістам – кожен займається своїм компонентом. При цьому розробник не повинен втручатися у чужий код. Його робота не впливає на інші фрагменти.

Шаблон MVC вирішує такі завдання:

- Зміна тільки інтерфейсу користувача, а не бізнес-логіки програми.
- Використання в одному додатку різних інтерфейсів із можливістю вибору.
- Заміна реакції програми на дії користувача за рахунок використання іншого контролера.

У більш широкому плані використання шаблону MVC допомагає:

- Спростити код великої програми, зробити його зрозумілим та структурованим, полегшити підтримку, тестування, повторне використання елементів.
- Організувати незалежну роботу різних відділів, що займаються розробкою своєї частини програмного продукту.
- Спростити програмну підтримку MVC-програми за рахунок модифікації окремих компонентів, а не всієї архітектури.

Використання MVC у веб-фреймворках на прикладі соціальної мережі.

Controller

Користувач заходить на сайт соціальної мережі та тисне на посилання «Друзі», надсилаючи запит на сервер.

website.com/profile/ —> повертає ваш профіль

website.com/friends/ —> повертає список друзів

website.com/friend={userName}/ —> повертає конкретного друга

Model

На сервері програма обробляє запит, витягує з бази даних список друзів користувача.

User:

{

userName: {

firstName,

lastName

},

friends

}

View

Інформація про друзів користувача виводиться на екран у вигляді списку з користувачами.

```
<ul>
    <li>Friend 1: {friendList[0].userName}</li>
    <li>Friend 2: {friendList[1].userName}</li>
    <li>Friend 3: {friendList[2].userName}</li>
</ul>
```

Концепція MVC з успіхом використовується у веб-розробці завдяки універсальності, гнучкості та простоті застосування. У той самий час вона має обмеження. Це призвело до появи більш спеціалізованих варіантів реалізації MVC, таких як Model-View-Presenter (MVP) та Model-View-ViewModel (MVVM).

MVP (Model-View-Presenter)

Шаблон MVP - один із найпопулярніших архітектурних шаблонів, що застосовують при організації проекту для мобільних пристроїв.

Використовуючи MVC як архітектуру програмного забезпечення, розробники стикаються з такими труднощами:

- Більшість основної бізнес-логіки перебуває у Controller. Протягом терміну служби програми цей файл стає більшим, і стає важко підтримувати код.
- Через тісно пов'язані механізми інтерфейсу користувача і доступу до даних рівень Controller і View потрапляють в одну і ту ж операцію або фрагмент. Це викликає проблеми при внесенні змін до функцій програми.
- Стає складно проводити модульне тестування різних рівнів, оскільки для більшої частини частин, що тестуються, потрібні компоненти Android SDK.

Шаблон MVP долає ці проблеми MVC та надає простий спосіб структурувати коди проекту. MVP набув широкого поширення, оскільки забезпечує модульність, зручне тестування, а також більш чисту та зручну в обслуговуванні кодову базу. Він складається з наступних трьох компонентів:

- Model. Це дані додатку, логіка їх отримання та збереження. Найчастіше базується на базі даних або результатах від веб-сервісів. У деяких випадках Model потрібно адаптувати, змінити або розширити перед використанням у View.
- View. Зазвичай є формою з віджетами. Користувач може взаємодіяти з його елементами, але коли якась подія віджету зачіпатиме логіку інтерфейсу, View направлятиме його до Presenter.
- Presenter. Містить всю логіку інтерфейсу користувача і відповідає за синхронізацію Model та View. Коли Model повідомляє Presenter, що користувач щось зробив (наприклад, натиснув кнопку), Presenter приймає рішення про оновлення Model та синхронізує всі зміни між Model та View. Presenter спілкується через інтерфейс і не спілкується з View безпосередньо. Завдяки цьому Presenter та Model можуть бути протестовані окремо.

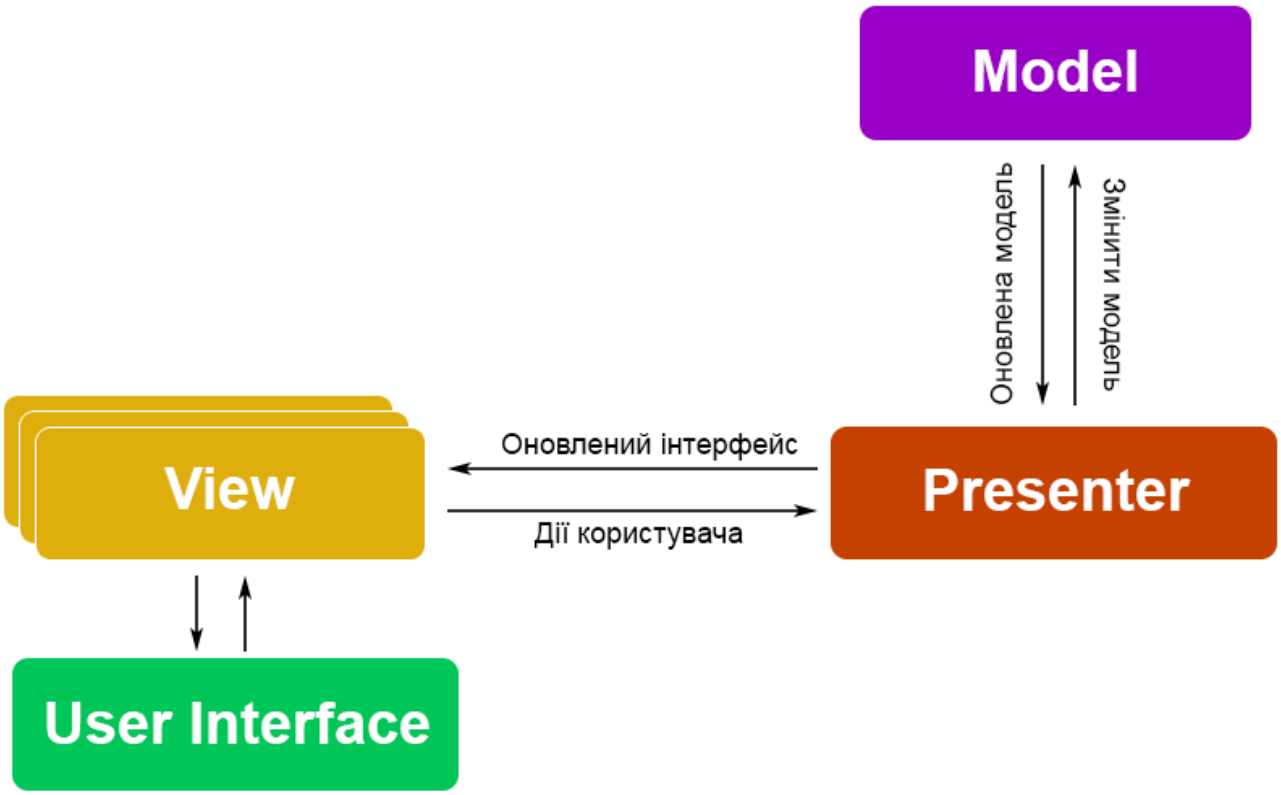

Рис. Логічна схема MVP-шаблону

У MVP-шаблоні View створює інтерфейс і підтягує дані з Model. Як тільки користувач робить будь-яку дію, View передає це, як подію, в Presenter. Presenter може сам обробити дані або одразу передати їх у Model. Model реагує виконанням відповідних операцій і віддає назад Presenter. Presenter передає оновлену Model у View і все починається спочатку.

Якщо в класичному MVC модель безпосередньо спілкується з View, то тут використовується посередник. Ось і вся різниця.

Ключові моменти архітектури MVP

- Зв'язок між View-Presenter та Presenter-Model відбувається через інтерфейс.
- Один клас Presenter одночасно управляє одним View, тобто між Presenter і View існує взаємно однозначний зв'язок.
- Класи Model та View не знають про існування один одного.

Шаблони MVC та MVP дуже схожі один на одного, але їх застосування залежить від умов використання. Для MVC – це там, де View оновлюється щоразу за якоюсь подією, а для MVP, коли View не потрібно щоразу перетворювати.

Переваги MVP

- **Зручний поділ.** Всі архітектури ПЗ призначені для того, щоб спростити та впорядкувати спільну розробку. Можна одночасно задіяти більше фахівців, які займатимуться окремими частинами програми. При цьому наявність шаблону гарантує чистоту та структурованість коду, що зменшує кількість помилок та спрощує тестування.
- **Перевірюваність.** Головна особливість MVP – наявність прошарку Presenter. Те, що дані передаються через Presenter, не ускладнює код, а навпаки, сприяє його стабільності. Presenter перевіряє стан та логіку View, що захищає програму від

критичних збоїв з випадкових причин. Крім того, ще на етапі тестування він допоможе швидше знаходити вузькі місця, а сама його наявність забезпечує простоту вирішення виявлених проблем.

- **Повторюваність.** Інтерфейс View краще за те, що використовується в класичному MVC тим, що може повторно використовувати бізнес-логіку, якщо Model не змінювалася. Це можливо завдяки жорсткому відділенню частин програми та гнучкого зв'язку через Presenter. Ця особливість дозволяє застосуванню бути менш вимогливим до ресурсів та більш гнучким.

MVC та MVP-парадигми дуже схожі один на одного, але їх застосування залежить від умов використання. Для MVC – це там, де View оновлюється щоразу за якоюсь подією, а для MVP, коли View не потрібно щоразу перетворювати.

MVVM (Model-View-ViewModel)

Недоліки шаблону MVP були вирішені за допомогою шаблону MVVM, який представив Android.

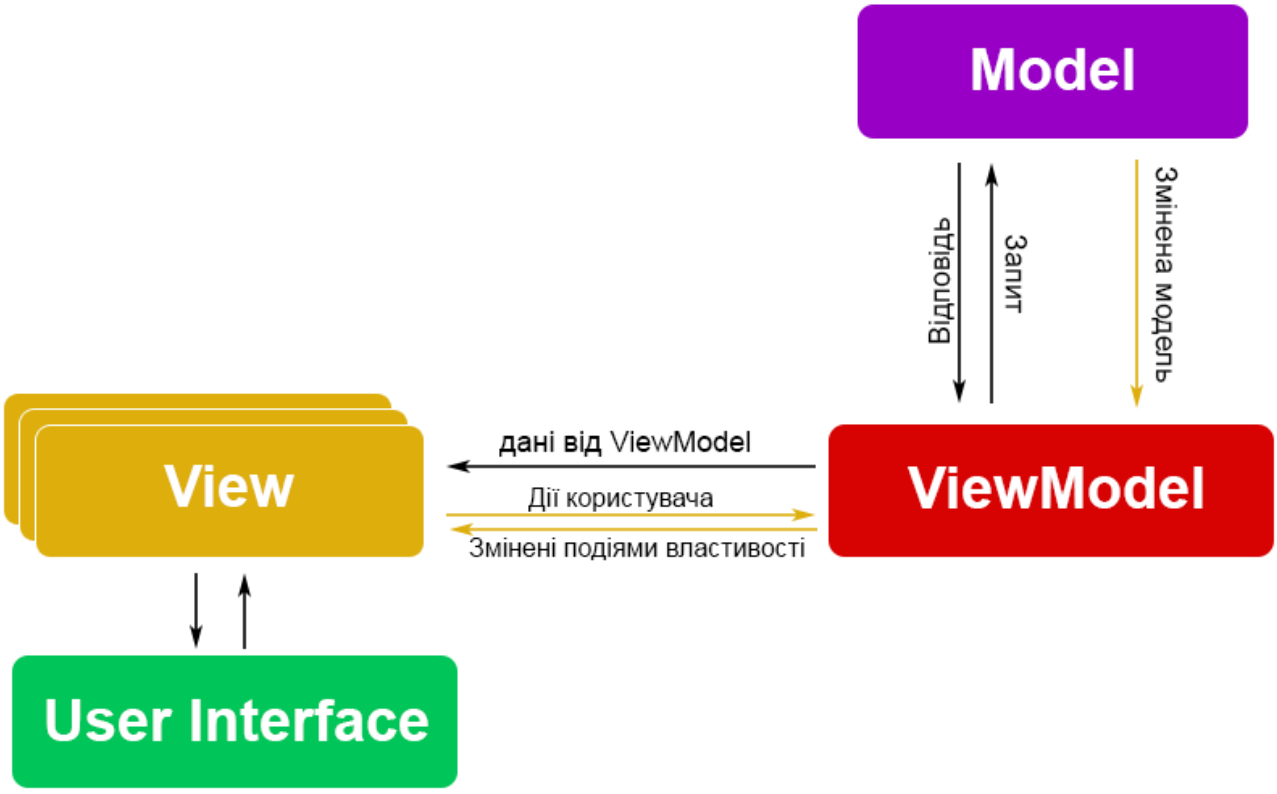

Рис. Логічна схема MVVM -шаблону

Шаблон MVVM не надто відходить від MVP. Компоненти View і Model практично такі ж, як і раніше, але ViewModel дещо відрізняється від Presenter:

- Model. Цей рівень відповідає за абстракцію джерел даних. Модель і ViewModel працюють разом, щоб отримати та зберегти дані.
- View. Мета цього шару — інформувати ViewModel про дії користувача. Цей рівень спостерігає за ViewModel і не містить жодної логіки програми.
- ViewModel: відкриває ті потоки даних, які мають відношення до View. Він служить сполучною ланкою між Model та View.

Різниця між шаблонами проектування MVC, MVP і MVVM

MVC	MVP	MVVM
Одна з найстаріших програмних архітектур	Розроблено як другу ітерацію архітектури програмного забезпечення, яка є передовою від MVC	Визнаний у галузі шаблон архітектури для додатків.
Інтерфейс користувача (View) і механізм доступу до даних (Model) тісно пов'язані між собою.	Вирішує проблему наявності залежного View, використовуючи Presenter як канал зв'язку між Model і View.	Більше керується подіями, оскільки використовує зв'язування даних і, таким чином, дозволяє легко відокремити основну бізнес-логіку від View.
Контролер і представлення існують із відношенням «один до багатьох». Один Controller може вибрати інше View на основі необхідної операції.	Між Presenter і View існує взаємозв'язок один-до-одного, оскільки один клас Presenter одночасно керує одним View.	Кілька представлень можна зіставити з однією ViewModel, і, отже, між View і ViewModel існує зв'язок «один до багатьох».
View не знає про Controller.	Перегляд має посилання на Presenter.	View має посилання на ViewModel
Важко вносити зміни та змінювати функції програми, оскільки рівні коду тісно пов'язані.	Рівні коду слабо пов'язані, тому легко вносити модифікації/зміни в код програми.	Легко вносити зміни в додаток. Однак, якщо логіка зв'язування даних надто складна, налагодити програму буде трохи важче.
Введення користувача обробляються Controller.	Перегляд є точкою входу до програми	View приймає вхідні дані від користувача та діє як точка входу програми.
Ідеально підходить тільки для невеликих проектів.	Ідеально підходить для простих і складних застосувань.	Не ідеально підходить для невеликих проектів.
Обмежена підтримка модульного тестування.	Модульне тестування легко проводити, але тісний зв'язок View і Presenter може дещо ускладнити його.	Тестування одиниць найвище в цій архітектурі.
Ця архітектура сильно залежить від Android API.	Він мало залежить від Android API.	Має низьку залежність або не залежить від Android API.
Він не дотримується принципу модульності та єдиної відповідальності.	Дотримується принципу модульності та єдиної відповідальності.	Дотримується принципу модульності та єдиної відповідальності.

Загалом, основна відмінність між цими шаблонами полягає в ролі посередницького компонента. MVC і MVP містять Controller або Presenter, який діє як посередник між Model і View, тоді як MVVM містить ViewModel, який служить посередником між Model і View. MVC є найпростішим із цих шаблонів, у той час як MVP і MVVM є більш гнучкими та дозволяють чіткіше розділяти завдання між різними рівнями програми.

Висновок

У сучасному висококонкурентному світі програмного забезпечення недостатньо створювати якісні продукти та послуги, щоб завоювати довіру клієнтів. Веб-додатки призначені, щоб якісно

надавати продукти та послуги користувачам. Таким чином, організаціям необхідно створювати та розгортати високооптимізовані веб-додатки, які економічно ефективно забезпечують швидкість і продуктивність, а також дружній інтерфейс користувача для клієнтів. Тут дуже важливо розробити правильну архітектуру веб-додатку.

Зрозумівши всі переваги та недоліки тієї чи іншої архітектури, можна вибрати доцільну архітектуру, якщо навіть мати обмежену інформацію щодо проекту. Будь-який проект можна виконати на будь-якій архітектурі, але це може бути неефективним або в плані розробки, або в плані експлуатації проекту.

Джерела інформації

1. Архітектура веб-додатків https://itanddigital.ru/webarchitecture
2. Концепції сучасної веб-архітектури https://skillbox.ru/media/code/top10-kontseptsiy-sovremennoy-vebarkhitektury-kotorye-vam-tochno-nuzhno-znat/
3. https://javarush.com/groups/posts/2519-chastjh-2-pogovorim-nemnogo-ob-arkhitekture-po