МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет комп'ютерних наук, фізики та математики
Кафедра інформатики, програмної інженерії та економічної кібернетики

РОЗРОБЛЕННЯ САЙТУ ПРАЦЕВЛТАШТУВАННЯ СТУДЕНТІВ ХДУ (BACK-END частина)

Кваліфікаційна робота

на здобуття ступеня вищої освіти «бакалавр»

Виконала: студент 4 курсу 441 групи
Спеціальності: 121 Інженерія
програмного забезпечення
Освітньо-професійної (наукової) програми:
Інженерія програмного забезпечення
Воропаєва Ірина Владиславівна
Керівники: доктор педагогічних наук,
професор Круглик Владислав Сергійович,
доктор педагогічних наук, професор,
Співаковський Олександр Володимирович
Рецензент: Федянін Павло Костянтинович, ФОП

Херсон – 2021

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА ТЕРМІНІВ

- **API –** програмний інтерфейс додатку
- **User story –** користувацька історія
- **ORM –** object-relational mapping, відображення бази даних на об’єкти об’єктно-орієнтованої мови програмування
- **EF –** Entity Framework, реалізація ORM з відкритим вихідним кодом для ASP.NET
- **Back-end –** серверна частина додатку
- **GDPR –** General Data Protection Regulation – загальний регламент з захисту персональних даних

ВСТУП

В сучасному світі безліч студентів поєднують навчання у вищому навчальному закладі з трудовою зайнятістю. «Раніше студенти вчилися і підробляли, а в нинішній час працюють і намагаються вчитися». Спостерігається така тенденція: сьогодні працюючі студенти більш залучені в їх трудову зайнятість, ніж в отримання знань. Причину цього ми бачимо в тому, що після закінчення вищого навчального закладу випускники стикаються з проблемами при працевлаштуванні. Випускникам все складніше побудувати свою кар'єру без початкового досвіду роботи, безпосередньо після закінчення ЗВО. Щоб уникнути таких проблем, багато студентів намагаються отримати такий досвід під час свого навчання [1].

Актуальність дослідження полягає в необхідності забезпечення студентів актуальною інформацією щодо можливості працевлаштування за професією, так чи інакше поєднаною з обраним фахом.

Метою роботи є формулювання вимог та розробка веб додатку для працевлаштування студентів та випускників Херсонського державного університету (ХДУ).

Об'єктом дослідження є технології створення веб додатків.

Предметом дослідження є розробка вимог та серверної частини веб додатку працевлаштування студентів.

Для реалізації мети було поставлено наступні завдання:

1. Аналіз існуючих додатків працевлаштування студентів та випускників
2. Вибір стеку технологій додатку
3. Формулювання вимог до додатку
4. Реалізація додатку

РОЗДІЛ 1. ОГЛЯД РЕСУРСІВ ПРАЦЕВЛАШТУВАННЯ

1.1. Аналіз існуючих додатків

Поширеними веб ресурсами для пошуку роботи, що стануть у нагоді студентам ЗВО та випускникам є work.ua, rabota.ua, olx.ua.

Work.ua має великий вибір роботодавців та постійно оновлювані вакансії на будь-який вибір. Перевагами цього веб ресурсу є простота використання та популярність, відсутність вузької спеціалізації серед професій. Недоліками цього ресурсу є велика кількість резюме, і резюме студента або випускника дуже легко може загубитися серед кандидатів з досвідом роботи, а також відсутність статистики працевлаштування за ЗВО.

Rabota.ua також має великий вибір роботодавців, окремий розділ працевлаштування для студентів, та простий користувацький інтерфейс. До недоліків також відноситься відсутність загальнодоступної статистики.

Olx.ua має окремий розділ для пошуку роботи, перевагами є велика кількість вакансій, але на цьому ресурсі вакансії майже не фільтруються зі сторони адміністрації та модерації веб-ресурсу, тому у кандидатів є ризик зіштовхнуся з недобросовісними роботодавцями.

Тобто, аналізуючи існуючі додатки, можна зробити висновки, що достатньо важко знайти універсальне рішення для пошуку роботи студентам та випускникам ХДУ.

1.2. Вибір стеку технологій

Для розробки back-end частини веб додатку працевлаштування сайту ХДУ в результаті проведеного аналізу, окремі частини якого представлені нижче, було прийнято ряд рішень щодо вибору технологій, в тому числі і таких, що суттєво впливають на архітектуру додатку.

Як результат, до основних технологій ввійшли нижче наведені.

- C# та його стандартні бібліотеки як основа стеку технологій;

- .NET Core 5 як платформа для побудови додатку;

- SQL Server як рішення для бази даних;

- .NET Core Web API – як фреймворк для розробки додатку;

- NuGet Package Manager – як менеджер пакетів та залежностей .NET;

- Entity Framework – як ORM для взаємодії з базою даних;

- Swagger як специфікація [2] та технологія документування API;

У якості системи контролю версій був використаний Git. Головною метою використання системи контролю версій є можливість у разі пошкодження значної частини програми повернути минулу версію. Нумерація версій релізів розроблюваного додатку виконана згідно зі специфікацією Semantic Versioning 2.0.0 [2].

РОЗДІЛ 2. ФОРМУЛЮВАННЯ ВИМОГ ДОДАТКУ

Маса проблем із програмним забезпеченням виникає через недосконаліть способів, які люди застосовують для збору, документування, узгодження і модифікації вимог до ПО. Проблеми можуть виникати через неформального збору інформації, передбачуваної функціональності, помилкових або неузгоджених припущень, недостатньо визначених вимог і безсистемного зміни процесу. Більшість людей при будівництві будинку навіть не цікавляться підрядниками, поки в повному обсязі не обговорять свої потреби і бажання і не уточнять всі деталі. Покупці розуміють, що внесення змін тягне за собою зміну ціни; їм це не подобається, але вони це розуміють. Однак люди весело шукають виправдання, коли справа стосується розробки ПО. [3]

Помилки, допущені на стадії збору вимог, складають від 40 до 60% всіх дефектів проекту (Davis, 1993; Leff ing well, 1997). Дві найбільш поширені проблеми, про які повідомляється в великому Європейському огляді індустрії ПЗ, - визначення і керування вимогами замовника (ESPITI, 1995). Тим не менше багато організацій ще застосовують неефективні методи на цих стадіях розробки ПЗ. Типовий результат - неочікувані прогалини, а також відмінності між тим, що розробники збираються будувати, і тим, у чому клієнти реально потребують. [3]

Для формулювання вимог до додатку працевлаштування студентів та випускників Херсонського державного університету було використано формат користувацьких історій.

2.1. Функціональні вимоги до додатку

Призначена для користувача історія (user story) містить опис функціональності, яка буде представляти цінність для користувача або покупця програмного продукту. Концепція користувальницької історії охоплює три аспекти: [4]

• текстовий опис історії, яке використовується при плануванні проекту і служить нагадуванням про те, що саме має бути зроблено;

• усні обговорення історії, що конкретизують її деталі;

• тести, які відображають і документують деталі історії і можуть застосовуватися для підтвердження того факту, що її розробка завершена.

З урахуванням того, що призначені для користувача історії зазвичай пишуться від руки на паперових картках, Рон Джеффріс запропонував для зазначеної тріади аспектів чудову алітерацію 3C: Card (картка), Conversation (обговорення), Confirmation (підтвердження) [4].

Будучи найбільш наочним втіленням користувальницької історії, картка аж ніяк не є найважливішим її компонентом. Як сказала Рейчел Дейвіс, картки лише "представляють вимоги замовника, але не документують їх". Суть концепції можна сформулювати так: паперова картка містить лише стислий опис користувальницької історії, тоді як деталі самої історії з'ясовуються в ході обговорення і фіксуються в підтвердженні. [4]

Для розробки додатку було обговорено та прийнято наступні користувацькі історії (з розподіленням на ролі). Перша роль – користувач, найбільш узагальнена. Користувачем є будь-який гість у системі, або людина, що увійшла під своїми даними (зв’язка логін-пароль).

1. Як користувач, я хочу мати можливість реєструватися для того, щоб мати можливість подальшої роботи з системою
2. Як користувач, я хочу мати можливість входити до системи із раніше створеними даними (зв’язка логін-пароль), для того, щоб отримати доступ до додавання нових сутностей (вакансії та\або резюме) у систему
3. Як користувач, я хочу мати можливість скидати пароль, для того, щоб бути у змозі відновити його, якщо я його забув
4. Як користувач, я хочу мати можливість виходити з системи для того, щоб потім мати можливість змінити аккаунт, під яким я увійшов (для цього мені потрібно ввести нові дані)

Друга важлива роль у системі – роботодавець – це особа або група осіб, що можуть розміщувати вакансії, переглядати резюме та відповідати на відгуки на раніше створені у системі вакансії. Сутність ролі роботодавця у розміщенні нових пропозицій роботи.

1. Як роботодавець, я хочу мати можливість додавати вакансію з інформацією про посаду та компанію, для того, щоб мати можливість знайти нових співробітників
2. Як роботодавець, я хочу мати можливість переглядати інші вакансії, щоб порівнювати свої умови з умовами у інших місцях
3. Як роботодавець, я хочу мати можливість відповідати на листи шукачів роботи для того, щоб мати із ними зв’язок для подальшої співбесіди та працевлаштування
4. Як роботодавець, я хочу мати можливість редагувати вакансію для того, щоб у випадку зміни вимог до вакансії, відобразити зміни у системі

5. Як роботодавець, я хочу мати можливість видаляти (деактивувати) вакансію для того, щоб у випадку, коли співробітник знайдений, не отримувати нових відгуків на неактуальну вакансію.
6. Як роботодавець, я хочу мати можливість пошуку, фільтрації та пагінації резюме для того, щоб швидше знаходити кандидатів, що задовольняють вимогам вакансії

Наступна роль у системі – шукач роботи. Це людина, що зареєструвалася у системі, як шукач, та має можливість відгукатися на вакансії.

1. Як шукач роботи, я хочу мати можливість додавати своє резюме, для того, щоб мати можливість відгукнутися на вакансію з його використанням
2. Як шукач роботи, я хочу мати можливість коригувати своє резюме, у випадку зміни моїх даних, що стосуються резюме
3. Як шукач роботи, я хочу мати можливість ховати (деактивувати) своє резюме для того, щоб резюме не залишалося активним у випадку, коли робота була знайдена
4. Як шукач роботи, я хочу мати можливість відправляти своє резюме вакансіям, що сподобались для того, щоб мати змогу зв’язатися с роботодавцем та обрати дату проведення співбесіди

Наступна наявна роль у системі – адміністратор. Ця роль має найширші права серед усіх, і має значний вплив на дані, що накопичуються у системі. Роль адміністратора є відповідальною, тому вона має бути лише у малої кількості осіб, що відповідають за систему.

1. Як адміністратор, я хочу мати можливість активувати та деактивувати користувачів будь-яких ролей, виключаючи інших адміністраторів для того, щоб у мене була можливість позбавити

доступу у систему невідповідальних або небезпечних користувачів

2. Як адміністратор, я хочу мати усі можливості, що мають інші ролі для того, щоб мати повноцінний доступ до системи
3. Як адміністратор, я хочу мати можливість додавати нових користувачів у систему із використанням Active Directory, що є частиною IT-інфрастуктури Херсонського державного університету для того, щоб надати доступ у відповідних ролях співробітникам університету
4. Як адміністратор, я хочу мати можливість змінювати ролі користувачів за їх вимогою та потребою для того, щоб підтримувати актуальний стан користувачів

Користувацькі історії – це функціональні вимоги до продукту, що розробляється, у даному випадку – сайту працевлаштування студентів та випускників Херсонського державного університету. Але, окрім функціональних вимог, у розробці програмного забезпечення існують й інші типи вимог, один з яких – нефункціональні вимоги.

2.2. Нефункціональні вимоги до додатку

Необхідно вказувати спеціальні вимоги до продуктивності для різних системних операцій. Також необхідно обґрунтовувати їх необхідність для того, щоб допомогти розробникам прийняти правильні рішення, що стосуються дизайну. Наприклад, через жорсткі вимоги до часу відгуку бази даних розробники можуть зеркалювати базу даних в декількох географічних місцях або денормалізувати пов'язані таблиці баз даних для отримання більш швидкої відповіді на запити. [3]

Для розроблюваного додатку було створено наступні вимоги до продуктивності:

1. Додаток повинен зберігати високу швидкість відповіді з API (не більше 3-х секунд на запит) при одночасному доступі 100 відвідувачів;
2. Додаток не повинен використовувати обсяг пам'яті, більший за 100 мегабайт при кількості 100 відвідувачів;
3. Єдиний сервер бази даних повинен швидко (не більше 5 секунд на запит) обслуговувати запити, навіть при наявності декількох (до 3-х) серверів додатків, що діють під високим навантаженням (більше 1000 користувачів одночасно).

Юридичною вимогою до додатку є слідування GDPR.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ BACK-END ЧАСТИНИ ДОДАТКУ

3.1. Вибір способу зберігання інформації

Зважаючи на те, що у додатку важлива цілісність даних, є необхідність швидкого додавання та виборки інформації, у якості рішення для збереження даних було обрано SQL Server. Він масштабується, тому працювати з нею можна на портативних ПК або потужної мультипроцесорної техніці. Процесор може одночасно обробляти великий обсяг запитів.

SQL Server - це сервер баз даних від Microsoft.

Система управління реляційними базами даних Microsoft - це програмний продукт, який в основному зберігає та отримує дані, запитувані іншими програмами. Ці програми можуть працювати на одному або іншому комп'ютері.

Поглиблюючись, щоб зрозуміти, що таке SQL Server, спочатку потрібно зрозуміти, що таке SQL.

SQL - це спеціальна мова програмування, призначена для обробки даних у реляційній системі управління базами даних. Сервер баз даних - це комп'ютерна програма, яка надає послуги баз даних іншим програмам або комп'ютерам, як визначено моделлю клієнт-сервер. Отже, SQL Server - це сервер баз даних, який реалізує структуровану мову запитів (SQL).

Існує багато різних версій Microsoft SQL Server, що відповідають різним навантаженням та вимогам. Версія центру обробки даних адаптована до вищих рівнів підтримки програм та масштабованості, тоді

як версія Express - це зменшена, безкоштовна версія програмного забезпечення.

Розмір сторінок - до 8 кб, тому дані витягуються швидко, детальну і складну інформацію зберігати зручніше. Система дозволяє обробляти транзакції в інтерактивному режимі, є динамічне блокування.

Рутинні адміністративні завдання автоматизовані: це управління блокуваннями, пам’яттю, редактура розмірів файлів. У системи продумані налаштування, можна створити профілі користувачів. Реалізовано пошук по фразах, тексту, словами, можна створювати ключові індекси. У SQL Server є реплікації через інтернет, передбачена синхронізація. Є повноцінний веб-асистент для форматування сторінок.[6]

В систему інтегрований сервер інтерактивного аналізу для прийняття рішень, створення корпоративних звітів. Є служби перетворення інформації.

Для менеджменту бази даних було обрано SQL Server Management Studio.

SQL Server Management Studio - це графічний інтерфейс, що входить до складу SQL Server 2005 і пізніших версій для налаштування, управління та адміністрування всіх компонентів у Microsoft SQL Server. Інструмент включає як редактори сценаріїв, так і графічні інструменти, що працюють з об’єктами та функціями сервера. [12]

SQL Server Management Studio також доступна для SQL Server Express Edition, для якої вона відома як SQL Server Management Studio Express (SSMSE). [13]

Центральною особливістю SQL Server Management Studio є Object Explorer, який дозволяє користувачеві переглядати, вибирати та діяти з будь-якими об'єктами на сервері. [14]

Він може використовуватися для візуального спостереження та аналізу планів запитів та оптимізації роботи бази даних, серед іншого. Студія управління SQL Server також може використовуватися для створення нової бази даних, зміни будь-якої існуючої схеми бази даних шляхом додавання або модифікації таблиць та індексів або аналізу продуктивності. Він включає вікна запитів, які забезпечують інтерфейс на основі графічного інтерфейсу для написання та виконання запитів. [15]

3.2. Реалізація бази даних

Для роботи з базою даних використаний Entity Framework – далі EF. Це зручний інструмент для взаємодії з базою даних з програмного коду, що надає широкий спектр можливостей для роботи з обраною базою даних - SQL Server.

Модель EF зберігає детальну інформацію про те, як класи та властивості програми відображаються у таблицях та стовпцях баз даних. Існує два основних способи створення моделі EF: [7]

Використання коду спочатку (code-first підхід): розробник пише код, щоб вказати модель. EF генерує моделі та відображення під час виконання на основі класів сутності та додаткової конфігурації моделі, наданої розробником. [7]

Використання EF Designer: Розробник малює рамки та лінії, щоб вказати модель за допомогою EF Designer. Отримана модель

зберігається як XML у файлі з розширенням EDMX. Доменні об'єкти програми, як правило, генеруються автоматично з концептуальної моделі.[7]

Для створення бази даних був використаний code-first підхід, тому що він є найбільш зручним для розробників програмного забезпечення, що мають досвід із платформою .NET, але не є розробниками баз даних (database developers – цей підхід не позбавляє можливості використання у майбутньому SQL збережуваних процедур та raw SQL («сирого» SQL, тобто прямих запитів до бази даних, пропускаючи шар абстракції, пов’язаний с Entity Framework).

Але для задач, пов’язаних з розробкою ресурсу для працевлаштування студентів та випускників Херсонського державного університету достатньо можливостей фреймворку EF та LINQ – language integrated query, розширення стандартного синтаксису C# для роботи з перелічуваними та запитуваними структурами даних (перші імплементують інтерфейс .NET – IEnumerable, другі – IQueryable). Для класів, що імлементують IQueryable у наявності є асинхронні імплементації методів.

Для оновлення структури бази даних та статичних даних у проєкті використовується механізм міграцій.

При додаванні або зміні нових сутностей або властивостей схеми бази даних повинні бути відповідним чином змінені для синхронізації з додатком. Функція міграції в EF Core дозволяє послідовно застосовувати зміни схеми до бази даних, щоб синхронізувати її з моделлю даних в додатку без втрати існуючих даних. [7]

Загалом міграції працюють таким чином. При появі зміни моделі даних розробник використовує EF, щоб додати відповідну міграцію з

описом оновлень, необхідних для синхронізації схеми бази даних. EF Core порівнює поточну модель з моментальним знімком старої моделі (так званий database snapshot, снапшот бази даних) для визначення відмінностей і створює вихідні файли міграції. [7]

Файли можна відстежувати в системі управління версіями проекту, як і будь-які інші вихідні файли. Створену міграцію можна застосовувати до бази даних різними способами. EF записує всі застосовані міграції в спеціальну таблицю журналу, з якої буде ясно, які міграції були застосовані, а які ні. [7]

У базі даних додатку працевлаштування студентів та випускників ХДУ використовується таблиця EFMigrationsHistory (рис. 1), що має у собі два поля – MigrationId (це поле є натуральним первинним ключем типу varchar – тобто строковий запис, що є назвою міграції, що була додана) та ProductVersion – версія продукту.

Рис 1. Таблиця з міграціями

Таблиці у базі даних пов’язані між собою зовнішніми ключами (foreign keys). Це дає можливість зберігати цілісні дані у різних таблицях. Для конфігурування дій з записами, що мають зовнішній ключ

на іншу таблицю, під час видалення записів з первинним ключем, на який посилається зовнішній, у міграції, у якій створюється ця таблиця, використовується передаваний у метод параметр onCascade – це булева змінна, що відповідає за те, чи видаляти каскадно записи за зовнішніми ключами.

Якщо цей флаг має значення false, при спробі видалити записи, на які є зовнішні посилання (зовнішні ключи) – додаток отримує помилку часу виконання (runtime exception), і записи не будуть видалені.

На рис. 2 зображена частина схеми бази даних, пов’язаної з вакансіями. Вакансії, що додаються, мають зовнішній ключ на id рекрутера, що додав цю вакансію, можливі зони роботи.

Термін «схема бази даних» може означати як наочне уявлення бази даних, так і набір правил, яким вона підпорядковується, або повний комплект об'єктів, що належать конкретному користувачеві. Схема бази даних являє собою логічну конфігурацію або цілої реляційної бази даних, або її частини. Схема може існувати як у вигляді наочного уявлення бази даних, так і у вигляді набору формул (також іменованих «умовами цілісності»), які регулюють її пристрій. Ці формули виражаються за допомогою мови опису даних, наприклад, SQL. Будучи частиною словника даних, схема показує, як пов'язані між собою сутності, з яких складається база даних (таблиці, представлення, збережені процедури і так далі). [9]

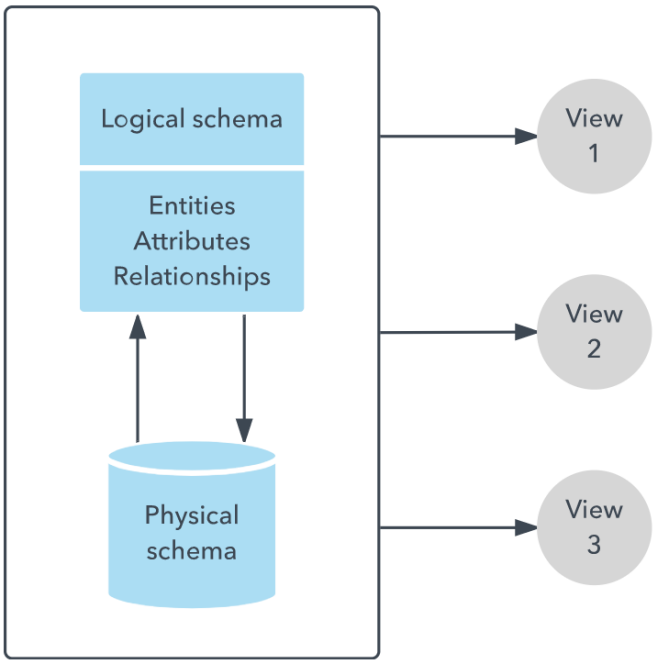

Рис 2. Узагальнене зображення схем баз даних

Процес створення схеми бази даних називається моделюванням даних. Якщо ви користуєтеся трехсхемним підходом до проектування бази даних, цей крок буде слідувати за створенням концептуальної схеми. Варто відзначити, що в центрі уваги концептуальної схеми знаходиться не структура бази даних, а інформаційні потреби організації. [9]

Виділяють два основних типи схем баз даних: *Логічна* схема бази даних демонструє логічні обмеження, які поширюються на збережені дані. У ній відображаються умови цілісності, уявлення і таблиці. *Фізична* база даних показує, як зберігаються дані в системі з точки зору файлів і індексів. [9]

Схема бази даних найпростішого рівня показує, з яких таблиць і зв'язків складається база даних, а також які поля входять до складу

кожної таблиці. Тому поняття «схема бази даних» та «схема» «сутність-зв'язок» часто взаємозамінні. [9]

Далі наведена схема бази даних, що використовується додатком для пошуку роботи студентів ХДУ.

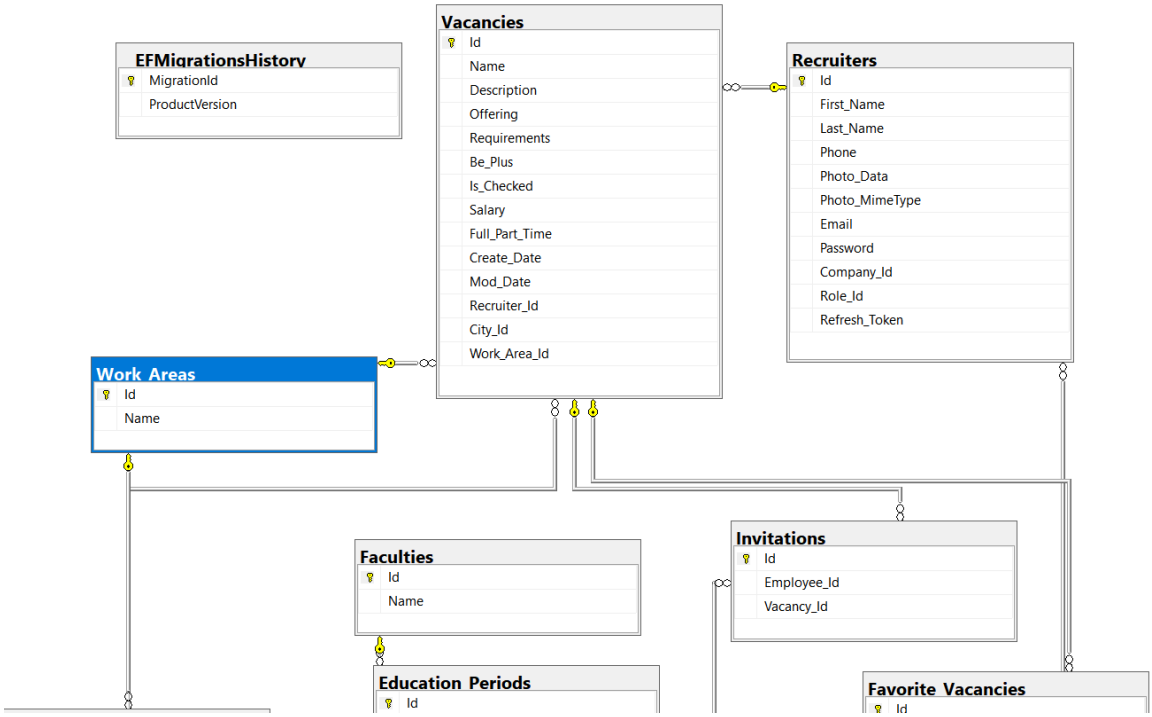

Рис 3. Таблиця вакансій та частина зовнішніх ключів на неї

На таблицю вакансій зовнішні ключи мають таблиця запрошень (це проміжна таблиця, для зв’язування таблиці найманого співробітника та вакансії). З точки зору коду таблиця запрошень описана наступним чином (рис. 3).

```
1. public class Invitation: Entity<int>
2.     {
3.         public override int Id { get; set; }
4.         public int EmployeeId { get; set; }
5.         public int VacancyId { get; set; }
6.
7.         public Employee Employee { get; set; }
8.         public Vacancy Vacancy { get; set; }
9.     }
10.
```

Рис 4. Таблиця запрошень з точки зору коду

Це один із можливих шляхів опису проміжних таблиць. Іншим шляхом опису проміжної таблиці є використання можливостей Entity Framework, наступним чином (рис. 4).

```
1. modelBuilder.Entity<Invitation>()
2.                 .HasMany(c => c.Employee)
3.                 .WithMany(s => s.Vacancy)
4.                 .UsingEntity(j => j.ToTable("Invitations"));
5.
```

Рис. 5. Таблиця запрошень із використанням будівельника моделей

За замовчуванням, Entity Framework використовує в якості імен таблиць бази даних назви класів моделі у множині, використовуючи правила англійської мови. Наприклад, клас Invitation відображається на таблицю Invitations, а клас Recruiter на таблицю Recruiters в базі даних. За допомогою анотацій даних або Fluent API можна явно вказати ім'я, що генерується таблиці в базі даних. [8]

Для явної установки імен таблиць в анотаціях використовується атрибут Table, якому передається ім'я таблиці.

Цей атрибут знаходиться в просторі імен System.ComponentModel. DataAnnotations.Schema, нижче показаний приклад його використання (рис. 5) [8]

```
1. [Table("Invitation")]
2. public class Invitation
3. {
4.     // ...
5. }
6.
7. [Table("Recruiter")]
8. public class Recruiter
9. {
10.    // ...
11. }
```

Рис. 6. Задання назви таблиці у базі даних явним чином

В T-SQL застосовується таке ж поняття схеми, як і в стандарті ANSI SQL. У стандарті SQL схема визначається як колекція об'єктів бази даних, що має одного власника і формує один простір імен. Простір імен (namespace) - це набір об'єктів з однозначними іменами. Наприклад, дві таблиці можуть мати одне і те ж ім'я тільки в тому випадку, якщо вони знаходяться в різних схемах. Схема є дуже важливим концептом в моделі безпеки компонента Database Engine СУБД SQL Server. Схемою за замовчуванням, для всіх об'єктів бази даних є схема dbo. [8]

Модель даних може містити класи, які не потрібно відображати у вигляді таблиць в базі даних. Навіть якщо ви явно не вкажіть ці класи в DbSet класу контексту, все одно можливі ситуації, коли Code-First спробує створити з них таблиці. Наприклад, як було сказано вище вказувати клас Order в DbSet не обов'язково, тому що клас Customer автоматично посилається на нього через навігаційне властивість. [8]

Що робити, якщо нам потрібно вказати Code-First, що клас Order не повинен відображатися в базі даних? Для цих цілей в анотаціях даних використовується спеціальний атрибут NotMapped, який застосовується до класу моделі. У Fluent API це реалізовано за допомогою методу Ignore() класу DbModelBuilder. [8]

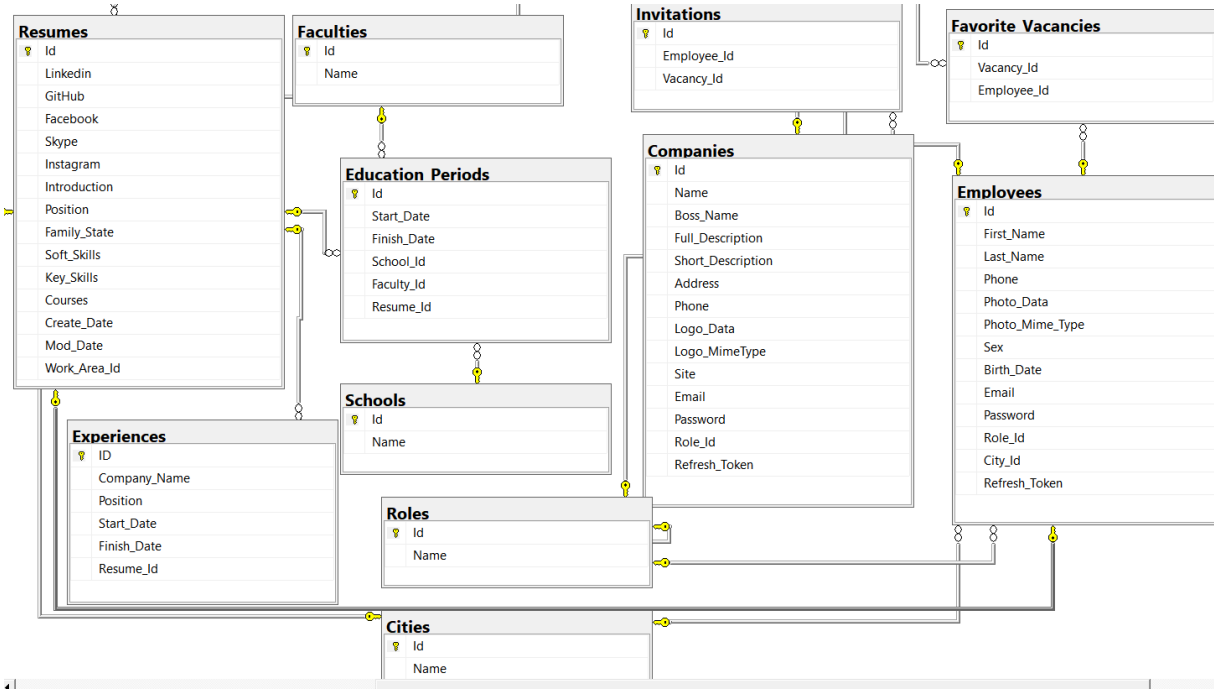

Рис. 7. Друга частина бази даних ресурсу

На рис. 6 зображена друга частина схеми бази даних розроблюваного ресурсу працевлаштування. На схемі присутні таблиця з резюме, зовнішній ключ якої використовується у таблицях з періодами навчання (Education_Periods), досвіді (Experiences), а також у таблиці мов, що перераховані у резюме (рис. 7).

Схема бази даних дає максимально детальне уявлення про те, як дані, що вводяться користувачами, використовуються у системі.

3.3. Шари додатку

Більша частина традиційних .NET додатків викладається у вигляді одного елементу, що відповідає виконуваному файлу, або одного веб додатку, що виконується у домені додатків та служб IIS. Це найпростіша модель розгортання, що є оптимальною для великої частини багатьох внутрішніх та невеликих загальнодоступних додатків. Для такої моделі розгортання більша частина бізнес-додатків використовує переваги логічного поділення на шари.[10]

Архітектура програми містить як мінімум один проект. У такому випадку вся логіка додатка укладена в одному проекті, компілюється в одну збірку і розгортається як один елемент.

У міру збільшення складності додатку для ефективного управління ним може застосовуватися розбиття по обов’язкам і завданням. Такий підхід відповідає принципу поділу завдань і допомагає зберегти можливість розширення кодової бази, завдяки чому розробники можуть швидко визначати, де саме реалізовані певні функції. Багатошарова архітектура має також цілий ряд інших переваг.[7]

Завдяки упорядкуванню коду за допомогою шарів загальні низькорівневі функції можуть багаторазово використовуватися по всьому додатку. Це вкрай важливо, оскільки такий підхід вимагає меншого обсягу коду і, за рахунок стандартизації додатку на рівні однієї реалізації, відповідає принципу "не повторювати" (з англійської DRY – “do not repeat yourself”).[11]

У додатках з багатошаровою архітектурою можуть встановлюватися обмеження на взаємодію між шарами. Таким чином вдається реалізувати принцип інкапсуляції. При зміні або заміні шару

будуть порушені тільки ті верстви, які працюють безпосередньо з ним. Обмежуючи залежності шарів один від одного, можна зменшити наслідки внесення змін, в результаті чого одиничне зміна не впливатиме на всі додаток.[7]

Застосування шарів (і інкапсуляція) дозволяє помітно спростити заміну функціональних можливостей в рамках програми. Наприклад, додаток може спочатку використовувати власну базу даних SQL Server для зберігання, а згодом перейти на стратегію збереження стану на основі хмари або веб-API. Якщо в додатку належним чином інкапсульована реалізація зберігання на логічному шарі, цей шар SQL Server може бути замінений новим, де буде реалізовуватися той же відкритий інтерфейс.[7]

Крім можливості заміни реалізацій в зв'язку з наступними змінами, застосування шарів в додатку також дозволяє змінювати реалізації з метою тестування. Замість написання тестів, які застосовуються до верств реальних даних або призначеного для користувача інтерфейсу додатку, під час тестування вони замінюються фіктивними реалізаціями, які демонструють відому реакцію на запити. Як правило, це значно спрощує написання і прискорює виконання тестів в порівнянні з тестуванням в реальній інфраструктурі додатку.[7]

Поділ на логічні шари широко поширене і допомагає впорядкувати код додатків. Зробити це можна кількома способами.[7]

Загальноприйнята організація логіки додатка по верствам показана на рис. 8.

Як правило, в додатку визначаються шари призначеного для користувача інтерфейсу, бізнес-логіки і доступу до даних. (рис. 9) В рамках такої архітектури користувачі виконують запити через шар

призначеного для користувача інтерфейсу, який взаємодіє тільки з шаром бізнес-логіки. Шар бізнес-логіки, в свою чергу, може викликати шар доступу до даних для обробки запитів. Шар призначеного для користувача інтерфейсу не повинен виконувати запити безпосередньо до шару доступу до даних і будь-якими іншими способами безпосередньо взаємодіяти з функціями зберігання.

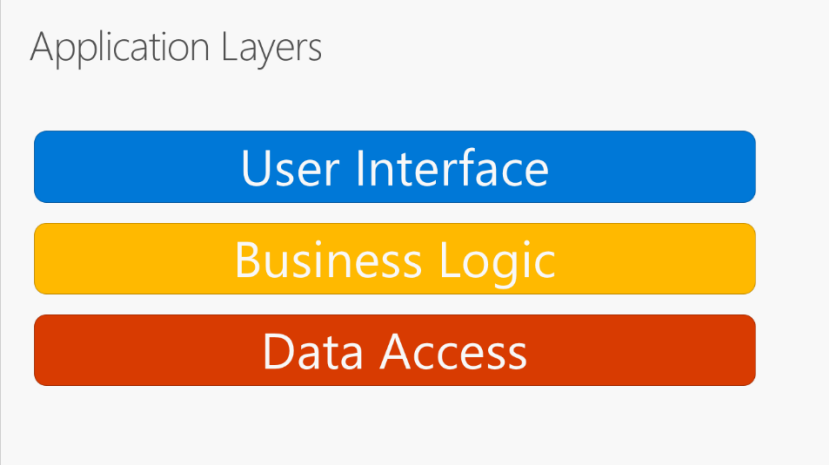

Рис. 8. Шари додатку у випадку N-шарової архітектури

Аналогічним чином, шар бізнес-логіки повинен взаємодіяти з функціями зберігання тільки через шар доступу до даних. Таким чином, для кожного шару чітко визначені свої обов’язки.[6]

Одним з недоліків традиційного багатошарового підходу є те, що обробка залежностей під час компіляції здійснюється зверху вниз. Це означає, що шар призначеного для користувача інтерфейсу залежить від шару бізнес-логіки, який, в свою чергу, залежить від шару доступу до даних. Також, шар бізнес-логіки, який зазвичай містить ключові функції програми, залежить від деталей реалізації доступу до даних (і часто від наявності самої бази даних). Тестування бізнес-логіки в такій архітектурі

часто утруднено і вимагає наявності тестової бази даних. Для вирішення цієї проблеми може застосовуватися принцип інверсії залежностей. [9]

На рис. 9 показаний приклад рішення, в якому додаток розділене на три проекти (або шару) відповідно до певних обов'язків.

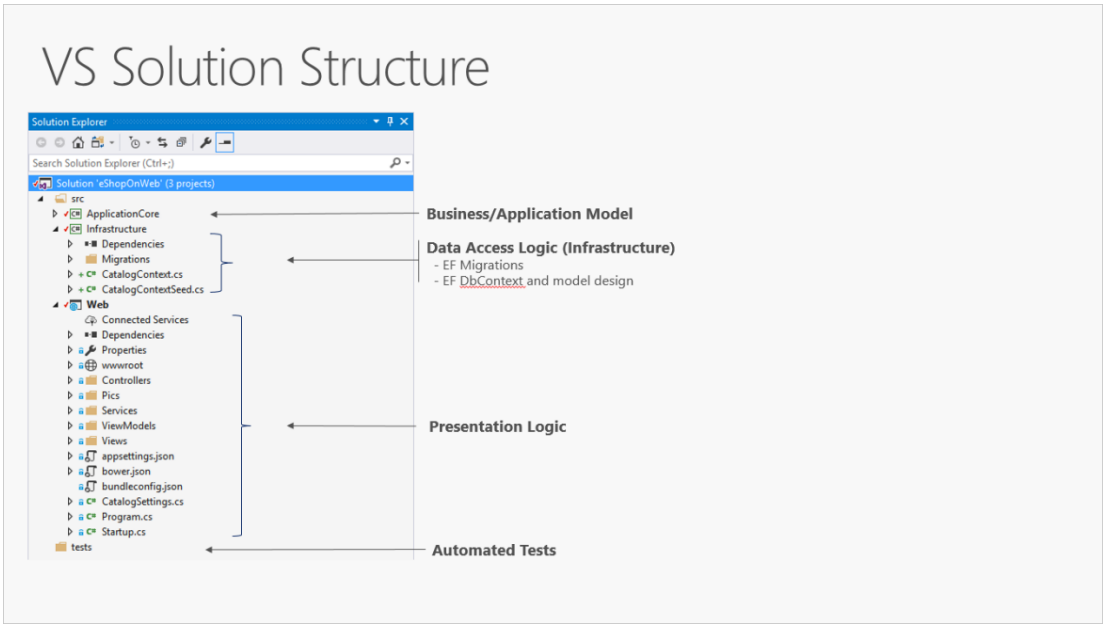

Рис. 9. Схема рішення у Visual Studio

Незважаючи на те, що з метою упорядкування в цьому додатку використовується кілька проектів, воно як і раніше розгортається як єдиний елемент, і його клієнти взаємодіють з ним як з одним веб-додатком. Це дозволяє реалізувати вкрай простий процес розгортання.

3.3.1. Шар доступу до даних

Для розробки додатку була обрана стандартна трьохшарова архітектура. Рішення ("solution") буде складатися з п'яти проєктів (рис. 10)

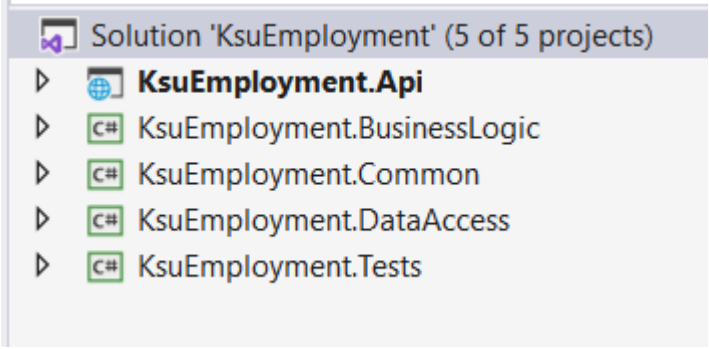

Рис. 10. Проєкти у рішенні додатку працевлаштування

Першим шаром, шаром доступу до даних є KsuEmployment.DataAccess – цей шар відповідає за доступ до бази даних з .NET коду створюваного додатку. На цьому шарі зберігаються сутності, що є відображенням таблиць у базі даних на класи C#, що традиційно звуться entities (сутності).

Також на цьому шарі знаходяться міграції проєкту, інтерфейси провайдерів (класів, що забезпечують доступ до даних із використанням сутностей) і спадкоємець класу DbContext, що використовується для зберігання усіх сутностей у якості властивостей класу, загорнутих у генералізований клас DbSet<SomeType>. Саме цей клас є цільним відображенням бази даних на код.

3.3.2. Шар бізнес-логіки

Наступним шаром додатку, згідно трьохшарової архітектури є шар бізнес-логіки. Особливість провайдерів у тому, що вони не мають зберігати ніякої додаткової логіки – лише виконувати операції CRUD – create, read, update, delete (додати, зачитати, оновити, видалити). Але у додатках, що подібні розроблюваному з’являється додаткова бізнес-логіка, така як валідація даних, додаткова їх обробка, тощо.

Для того, щоб виконати цю логіку можна використовувати сервіси – другий шар у трьохшаровій архітектурі. Сервіси не мають прямого доступу до бази даних, але мають доступ до провайдерів – це досягається за допомогою принципу розробки dependency injection - ін’єкція залежностей. У .NET це можна досягти за допомогою різних фреймворків, одним із найпопулярніший з яких це Autofac.

У кожного провайдеру має бути інтерфейс, який забезпечить публічний контракт того, що має робити провайдер. За допомогою Autofac ми маємо можливість пов’язати інтерфейс з його конкретною імплементацією (адже один інтерфейс можуть імплементувати декілька нащадків).

Наступним кроком є ін’єкція необхідної залежності. Є декілька шляхів – через метод, через конструктор або через властивість. Одним з найбільш поширених шляхів є ін’єкція через конструктор. Це корисно, адже це дозволяє у майбутньому легко розширювати кількість класів, що необхідно ін’єктувати (або навпаки – звузити їх кількість), а також спрощує юніт-тестування, адже дозволяє використати замість справжніх імплементацій інтерфейсу – фейкові, несправжні. Таким чином можна уникнути необхідності у юніт-тестах походів у базу даних, адже фейкова імплементація може або зберігати всі необхідні дані у пам’яті, або деінде.

Методи сервісів на цьому шарі також будуть містити CRUD операції, але крім прямого доступу до провайдера та пошуку даних у базі даних вони можуть, наприклад, шукати дані у кеші. Це дозволяє значно прискорити пошук у базі даних. Також саме тут має зберігатися усі особливості поведінки додатку, усі розрахунки та перевірки на коректність введених даних.

Прийнято на цьому шарі створювати окремі класи для переносу даних – data transfer objects, або dto – для того щоб відділити логіку сервісів від логіки шару даних. Наприклад, частою є ситуація, коли dto тримає у собі додаткові властивості, які не має необхідності зберігати у базі даних, але вони необхідні для того, щоб, наприклад, повертати їх у контроллер, або для інкапсулювання деякої логіки та запобігання дублювання коду.

3.3.3. Шар публічного інтерфейсу (API)

Останнім шаром трьохшарової архітектури, що буде розроблений у додатку буде розроблено шар API – application programming interface. У додатку використовується .NET Web API саме для того, щоб цей застосунок міг давати публічний інтерфейс доступу, який буде використаний фронт-енд частиною додатку.

На цьому рівні відбувається авторизація та аутентифікація користувача, перевірка його прав доступу. Саме на цьому рівні зберігаються кінцеві точки (end points) додатку, із різними методами – post, put, get, patch, delete.

Контролери мають бути асинхронними – виходячи з цього усі методи попередніх шарів також мають бути асинхронними, що забезпечить більш ефективне використання ресурсів різними частинами достатку. Вони мають доступ до сервісів, але не мають мати доступу до провайдерів – для того, щоб відповідати принципам єдиної відповідальності та інкапсуляції логіки у різних частинах додатку.

У контролерах відбувається ін’єкція сервісів через конструктор – за таким самим принципом як і у сервісах. Це також полегшує тестування контроллерів.

3.4 Покриття додатку тестами

Unit-тестування (тестування модулів) - це тип тестування програмного забезпечення, де тестуються окремі блоки або компоненти програмного забезпечення. Метою є перевірити, що кожна одиниця програмного коду працює належним чином. Тестування модулів проводиться під час розробки (фази кодування) програми розробниками. Модульні тести ізолюють розділ коду та перевіряють його правильність. Одиницею може бути окрема функція, метод, процедура, модуль або об’єкт. [16]

Модульне тестування важливо, оскільки розробники програмного забезпечення іноді намагаються економити час, виконуючи мінімальне модульне тестування, і це міф, оскільки невідповідне модульне тестування призводить до дорогого виправлення дефектів під час тестування системи, інтеграційного тестування та навіть бета-тестування після побудови програми. Якщо належне тестування модулів проводиться на початку розробки, це в кінцевому підсумку економить час і гроші. [16]

Модульні тести допомагають виправити помилки на початку циклу розробки та заощадити витрати.

Це допомагає розробникам зрозуміти базу коду тестування та дозволяє швидко вносити зміни

Хороші модульні тести служать проектною документацією.

Модульні тести допомагають повторно використовувати код. Перенесіть як свій код, так і свої тести на новий проект. Налаштовуйте код, поки тести не запустяться знову. [16]

Для того, щоб провести модульне тестування, пишеться розділ коду для тестування певної функції в програмному застосунку. Розробники також можуть ізолювати цю функцію для більш ретельного тестування, яке виявляє непотрібні залежності між функцією, що перевіряється, та іншими блоками, щоб залежності можна було усунути. Зазвичай розробники використовують фреймворки для розробки автоматизованих тестових кейсів для модульного тестування. [16]

Unit-тестування буває двох видів:

1. Ручне
2. Автоматизоване

Модульне тестування зазвичай автоматизоване, але все ще може виконуватися вручну. Програмна інженерія не надає перевагу одному над іншим, але автоматизація є кращою. Ручний підхід до модульного тестування може використовувати покроковий інструкційний документ. [16]

За автоматизованого підходу - розробник пише розділ коду в програмі лише для тестування функції. Пізніше вони коментуватимуть і нарешті видалятимуть тестовий код під час розгортання програми. [16]

Розробник також може ізолювати функцію, щоб перевірити її більш ретельно. Це більш ретельна практика модульного тестування, яка передбачає копіювання та вставлення коду у власне середовище тестування, ніж його природне середовище. Виділення коду допомагає виявити непотрібні залежності між тестуваним кодом та іншими

одиницями або просторами даних у продукті. Потім ці залежності можна усунути. [16]

Зазвичай використовує фреймворк для розробки автоматизованих тестових кейсів. Використовуючи систему автоматизації, розробник кодує критерії в тесті, щоб перевірити правильність коду. Під час виконання тестових кейсів фреймворк реєструє невдалі тестові кейси. Багато фреймворків також автоматично повідомлятимуть про ці невдалі тестові випадки та повідомлятимуть їх. Залежно від тяжкості несправності, фреймворк може зупинити подальше тестування. [16]

Робочий процес модульного тестування:

1. Створення тестових справ
2. Перегляд / переробка
3. Виконання тестів.

Переваги модульного тестування

1. Розробники, які хочуть дізнатись, яку функціональність забезпечує програмна одиниця та як її використовувати, можуть ознайомитись з модульними тестами, щоб отримати базове розуміння API модуля. [16]
2. Модульне тестування дозволяє програмісту здійснити рефакторинг коду пізніше і переконатися, що модуль все ще працює коректно (тобто тестування регресії). Процедура полягає у написанні тестових випадків для всіх функцій та методів, щоб кожен раз, коли зміна спричиняє несправність, її можна було швидко виявити та виправити. [16]

3. Через модульний характер модульного тестування ми можемо тестувати частини проекту, не чекаючи завершення інших.

Недоліки модульного тестування

1. Не можна очікувати, що модульне тестування виявить кожну помилку програми. Неможливо оцінити всі шляхи виконання навіть у найбільш тривіальних програмах[16]
2. Модульне тестування за своєю суттю фокусується на одиниці коду. Отже, він не може виявити помилки інтеграції або широкі помилки системного рівня.
3. Рекомендується застосовувати модульне тестування разом із іншими тестовими заходами.

Найкращі практики модульного тестування

1. Тестові кейси повинні бути незалежними. У разі будь-яких покращень або зміни вимог, це не повинно впливати на тестові випадки. [16]
2. Перевіряти лише один код за раз.
3. Дотримуйтесь чітких та послідовних правил іменування для своїх модульних тестів[16]
4. У разі зміни коду в будь-якому модулі, переконайтеся, що для модуля є відповідний блок тестів, і модуль проходить тести перед зміною реалізації[16]
5. Помилки, виявлені під час модульного тестування, повинні бути виправлені перед переходом до наступного етапу в SDLC[16]
6. Прийміть підхід "тест як ваш код". Чим більше коду ви пишете без тестування, тим більше шляхів вам доведеться перевірити на наявність помилок. [16]

Unit-тестування визначається як тип тестування програмного забезпечення, де тестуються окремі блоки або компоненти програмного забезпечення.

Модульне тестування може бути складним або досить простим залежно від програми, що тестується, та стратегій тестування, інструментів та філософії, що використовуються. Модульне тестування завжди потрібно на якомусь рівні. Це певність.

Розроблюваний додаток було покрито unit-тестами із використанням фреймворку MSTest.

ВИСНОВКИ

На основі проведеного аналізу визначено функціонал додатку та створені функціональні та нефункціональні вимоги із використанням способу опису вимог у вигляді користувацьких історій.

Для виконання поставлених задач було проаналізовано існуючі додатки пошуку роботи та працевлаштування. У якості технологій для розробки було обрано стек технологій – мова програмування C# та фреймворк розробки кросплатформених веб додатків .NET Core.

Було створено реляційну базу даних SQL Server та розроблена база даних. Було обрано стандартну трьохшарову архітектуру, що складається с шару інтерфейсу, бізнес-логіки да шару доступу до даних у якості архітектури, що використовується для розробки додатку, та ця архітектура на верхньому рівні була імплементована як рішення з п’ятьма проєктами.

ДОДАТОК А. КОДЕКС АКАДЕМІЧНОЇ ДОБРОЧЕСНОСТІ

Завідувачу кафедри
інформатики, програмної інженерії
та економічної кібернетики
Володимиру ПЕСЧАНЕНКУ
здобувача вищої освіти
Воропаєва Ірина Владиславівна
(ПІПб)
освітньо-професійної програми:
Інженерія програмного забезпечення
441 групи (денна ф.н.)

ЗАЯВА

Я, Воропаєва Ірина Владиславівна, підготував (ла) кваліфікаційну роботу (проєкт) на тему «Розроблення сайту працевлаштування студенів ХДУ (back-end частина)» особисто (з урахуванням внеску наукового керівника).

З правилами чинного Порядку про виявлення та запобігання академічного плагіату у науково-дослідній та навчальній діяльності здобувачів вищої освіти, згідно з яким виявлення плагіату є підставою для відмови в допуску роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а).

Про використання системи UNICHECK для виявлення текстових збігів/ідентичності/схожості в роботах здобувачів вищої освіти ознайомлений (а) та надаю Університету право на передачу моєї роботи для обробки та збереження в системі виявлення текстових збігів/ідентичності/схожості та використання роботи для виявлення плагіату в інших роботах, які завантажувалися/завантажуються/завантажуватимуться для перевірки системою виявлення текстових збігів/ідентичності/схожості та користувачами, які мають доступ до цієї системи, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки Університетом надається в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

Дата 09.04.2021

Підпис

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Кіберленінка. URL: https://cyberleninka.ru/article/n/rabotayuschiy-student (дата звертання 18.03.2021)
2. Semantic Versioning 2.0.0. URL: http://semver.org (дата звернення: 25.03.2021)
3. К. Вігерс: Розробка вимог до програмного забезпечення. Стереотипне 3 видання. БХВ-Петербург, 2014. 575 с.
4. Р. Джефріс: Extreme Programming Adventures in C#, O'Reilly, 2004, 560 с.
5. Intersoft Consulting. GDPR. URL: https://gdpr-info.eu/ (дата звертання 29.03.2021)
6. SQL Server 2019. URL: https://www.microsoft.com/ru-ru/sql-server/sql-server-2019 (дата звертання: 29.03.2021)
7. Microsoft Docs. URL: https://docs.microsoft.com/ (дата звертання 30.03.2021)
8. ProfWeb. URL: https://professorweb.ru/ (дата звертання 30.03.2021)
9. LucidChart. URL: https://www.lucidchart.com/ (дата звертання 1.04.2021)
10. Лаврищева Катерина Михаліївна : Програмна інженерія. Київ, 2008. 319 с.
11. Еммерих Вольфган Конструювання розподілених об'єктів. Методи та властивості програмування інтероперабельних об'єктів в архітектурах OMG/CORBA, Microsoft COM и Java RMI. Київ, 2002. 510 с.
12. MSDN: Introducing SQL Server Management Studio. URL: https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?redirectedfrom=MSDN&view=sql-server-ver15 (дата звертання 30.03.2021)

13. SQL Server Management Studio Express. URL: https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?redirectedfrom=MSDN&view=sql-server-ver15 (дата звертання 2.04.2021)
14. MSDN: Using Object Explorer. URL: https://docs.microsoft.com/en-us/sql/ssms/object/object-explorer?redirectedfrom=MSDN&view=sql-server-ver15 (дата звертання 2.04.2021)
15. SQL Server 2005 Management Tools. URL: http://www.sqlmag.com/ (дата звертання 3.04.2021)
16. Guru99. URL: https://www.guru99.com/ (дата звертання 5.04.2021)