МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «**Проєктування сайту для церкви**»

Виконав: студент 4 курсу, групи КН-4__
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Мосійчук олександр Ігорович

Керівник: *викладач кафедри ІТАД,*
Сергій Васильович Ляховчук

Рецензент: *кандидат технічних наук, доцент,*
доцент кафедри прикладної математики
Донецького національного університету
імені Василя Стуса
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних
______________ (проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від « 20 » травня 2026 р.

Острог, 2026

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема:Проєктування сайту для церкви

Автор: *Мосійчук Олександр Ігорович*

Науковий керівник: *Сергій Васильович Ляховчук , старший викладач*

Захищена «………»………………… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *50 с., 4 рис., 1 табл., 2 додатків, 40 джерел.*

Ключові слова: *FRONTEND, BACKEND, REACT, C#, .NET, ЦЕРКВА, ГРОМАДА, УПРАВЛІННЯ ПОДІЯМИ, ВЕБЗАСТОСУНОК, API, АВТОМАТИЗАЦІЯ, КОМУНІКАЦІЯ, КОНТРОЛЬ ДОНАЦІЙ*

Короткий зміст праці:

У кваліфікаційній роботі розглянуто процес проєктування та розробки повнофункціонального веб-сервісу для церковної громади, спрямованого на управління діяльністю, комунікацією з прихожанами та організацією заходів. Актуальність теми зумовлена необхідністю підвищення ефективності роботи церковних організацій, автоматизації обробки інформації та покращення взаємодії з членами громади через сучасні цифрові технології.

У роботі проаналізовано сучасні підходи до розробки вебзастосунків для релігійних організацій, визначено вимоги до користувацького інтерфейсу та серверної логіки, обґрунтовано вибір технологічного стеку. Система побудована за архітектурою клієнт-сервер з розділенням відповідальності: Frontend розроблено на React, Vite, Redux Toolkit та React-Bootstrap, Backend реалізовано на C# з використанням .NET платформи. Архітектура застосунку забезпечує модульність, масштабованість, надійність та безпеку даних.

Frontend частина передбачає розробку користувацького інтерфейсу з сучасним дизайном та інтуїтивною навігацією. Backend частина реалізує RESTful API для управління ресурсами, обробки запитів від клієнта, валідації даних та

взаємодії з базою даних.

Практична частина присвячена реалізації вебзастосунку для церковної громади. На фронтенді розроблено: сторінку про церкву, каталог подій, форму контактів, систему донацій, профілі користувачів, керування новинами та адміністративну панель. На бекенді реалізовано: управління користувачами, аутентифікацію та авторизацію, керування подіями, новинами, повідомленнями та донаціями. Передбачено механізми автентифікації, перевірки коректності даних, системи сповіщень та логування для ефективної комунікації та контролю системи.

У застосунку реалізовано режим тестування з mock-даними, інтеграційні тести та розгортання демо-версії. Frontend розгорнуто на платформі Vercel, Backend на хмарній платформі. У результаті створено повнофункціональний веб-сервіс, який дозволяє оптимізувати управління діяльністю церковної громади, автоматизувати основні процеси, скоротити час обробки інформації та значно покращити взаємодію та комунікацію з прихожанами.

This qualification work examines the design and development of a comprehensive web service for church community management, including both frontend and backend components. The relevance of the topic is determined by the need to improve the efficiency of church organizations, automate information processing, and enhance communication with community members through modern digital technologies.

The study analyzes modern approaches to web application development for religious organizations and justifies the selection of technologies. The system is built using a client-server architecture: Frontend is developed with React, Vite, Redux Toolkit, and React-Bootstrap, while Backend is implemented using C# and the .NET platform. The application architecture ensures modularity, scalability, reliability, and data security.

The Frontend component includes user interface design with modern aesthetics and intuitive navigation. The Backend component implements RESTful API for resource management, request processing, data validation, and database interaction.

The practical part describes the implementation of a comprehensive church community management web application. Frontend features include: church information pages, event catalog, contact forms, donation system, user profiles, news management, and administrative dashboard. Backend implements: user management, authentication and authorization, event management, news management, messaging, and donation processing. Mechanisms for data validation, notification systems, and comprehensive logging were implemented for effective communication and system control.

The application includes testing mode with mock data, integration tests, and demo deployment. Frontend is deployed on Vercel platform, Backend on cloud infrastructure. As a result, a comprehensive web service was created that optimizes church community management, automates core processes, reduces information processing time, and significantly improves interaction and communication with parishioners.

ВСТУП

В епоху тотальної цифровізації наявність зручного веб-ресурсу є важливою не лише для комерційного сектору, а й для громадських та некомерційних організацій. Сучасні користувачі прагнуть швидко знаходити потрібну інформацію, тому потреба в інтуїтивно зрозумілих онлайн-платформах невпинно збільшується. Для сучасної церкви надійне веб-представництво — це необхідна умова для ефективної комунікації з аудиторією. Налагоджена система онлайн-взаємодії допомагає не лише своєчасно інформувати про діяльність, але й суттєво підвищує залученість людей до життя громади.

Створення Frontend-складової веб-сайту для організації є нагальним завданням, адже саме сучасний інтерфейс гарантує зручний доступ до інформації та безперебійну комунікацію. Від якості клієнтської частини безпосередньо залежить користувацький досвід (UX): продумана архітектура та приваблива візуальна оболонка допомагають відвідувачу швидко знайти розклад подій, новини чи відповіді на запитання. Застосування передових технологій робить цей процес безпечним, інтерактивним та швидким.

Мета роботи полягає у розробці клієнтського боку веб-платформи, який надасть користувачам можливість зручно ознайомлюватися з діяльністю організації, переглядати анонси подій, робити благодійні внески та отримувати актуальні інформаційні матеріали. Окремий акцент робиться на швидкодії системи та її адаптивності, що гарантує однаковий комфорт при користуванні як з персональних комп'ютерів, так і з мобільних пристроїв.

Для досягнення поставленої мети було обрано такий стек сучасних технологій:

- ReactJS — як базова бібліотека для побудови архітектури на основі компонентів;
- Redux Toolkit — для централізованого управління станами застосунку (наприклад, управління сесією користувача, статус обробки платежів чи завантаження розкладу);
- Vite — як високошвидкісний інструмент для збирання проєкту

React-Bootstrap та SCSS — для розробки гнучкого, адаптивного та візуально привабливого дизайну.

Основні завдання, які необхідно вирішити в процесі роботи:

1. Аналіз наявних програмних рішень для веб-представництв подібних спільнот та організацій;
2. Проєктування архітектурних рішень фронтенду та моделі даних;
3. Програмування функціоналу для перегляду новин, розкладу подій та, за необхідності, створення особистого кабінету користувача;
4. Інтеграція безпечних інструментів для здійснення благодійних внесків та форм зворотного зв'язку;
5. Забезпечення високих стандартів UI/UX завдяки використанню адаптивного веб-дизайну.

Об’єктом дослідження виступає процес інформаційної взаємодії користувача з діяльністю організації через веб-інтерфейс.

Предметом дослідження є інструментарій та методи розробки Frontend-частини веб-ресурсу з використанням бібліотеки ReactJS та пов'язаних технологій управління станом.

Впровадження даного проєкту дасть змогу організації автоматизувати інформування аудиторії, оптимізувати комунікаційні процеси та запропонувати сучасний сервіс для цифрової взаємодії. Практичне значення роботи зводиться до створення готового до експлуатації інтерфейсу, що гармонійно поєднує візуальну естетику, необхідний функціонал та відмінну продуктивність.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ТА АНАЛІТИЧНІ АСПЕКТИ СТВОРЕННЯ ВЕБ-ОРІЄНТОВАНОЇ СИСТЕМИ ДЛЯ РЕЛІГІЙНОЇ ОРГАНІЗАЦІЇ

1.1. Постановка проблеми, мета та завдання

Процес розробки сучасних комплексних вебзастосунків є багатогранним та ітеративним завданням, яке вимагає не лише глибоких технічних знань, але й ґрунтовного попереднього аналізу, а також чіткого структурування існуючої проблематики предметної області. У сучасному глобалізованому інформаційному суспільстві, яке характеризується безпрецедентними темпами технологічного розвитку, цифрова трансформація невідворотно охоплює всі без винятку сфери суспільного та індивідуального життя. Це стимулює парадигмальний перехід від традиційних, часто застарілих методів міжособистісної та інституційної комунікації до інтерактивних, високошвидкісних та багатофункціональних високотехнологічних платформ. Створення спеціалізованого програмного забезпечення для потреб громадських та релігійних організацій постає як надзвичайно складне інженерне та соціально-технічне завдання. Воно вимагає від команди розробників не лише написання програмного коду, але й гармонійного поєднання передових інформаційних технологій, глибоко продуманого, емпатичного користувацького інтерфейсу та абсолютно надійних механізмів криптографічного захисту й збереження персональних даних.

1.1.1. Актуальність теми, мета та завдання дослідження

В епоху тотальної глобальної діджиталізації та повсюдного проникнення інтернету в повсякденну рутину, наявність єдиної, функціонально повноцінної цифрової екосистеми стає критично важливою умовою для підтримки стабільного, безперервного та якісного зв'язку між будь-якою організацією та її членами. Актуальність даного кваліфікаційного дослідження безпосередньо зумовлена об'єктивною, продиктованою часом необхідністю впровадження сучасних

інноваційних інформаційних інструментів в оперативну діяльність церковної громади. Традиційні засоби сповіщення та управління не здатні повною мірою задовольнити потреби сучасного користувача, який очікує миттєвого доступу до інформації з будь-якої точки світу.

Головна мета розробки клієнтської частини (фронтенду) полягає у тому, щоб на концептуальному та технічному рівнях забезпечити максимально зручний, ергономічний інтерфейс для кінцевих користувачів та адміністративного персоналу церковної громади. Окрім простої демонстрації статичного контенту, цей інтерфейс має виконувати складну роль посередника, тобто обробляти двосторонню взаємодію з REST API серверного бекенду та динамічно візуалізувати отримані масиви даних у зрозумілому для людини форматі. До критично важливих даних, які підлягають візуалізації та обробці, належать розклади подій, інформаційні новини, транзакції донацій та персональні профілі користувачів.

Для повноцінного досягнення цієї комплексної мети в рамках повного циклу розробки серверної (Backend) та клієнтської (Frontend) частин проєкту необхідно послідовно виконати низку взаємопов'язаних інженерних завдань. Серверна частина системи проєктується та розробляється як потужний ASP.NET Core API проєкт, який бере на себе всю тягарі бізнес-логіки та безпосередньо реалізує централізоване управління профілями користувачів, стрічками новин, календарями подій, вхідними повідомленнями та загальною статичною інформацією про церкву. Відповідно до цього, клієнтська частина повинна виступати бездоганним відображенням серверної логіки та забезпечити безперебійний, безпечний доступ до всього спектру цього функціоналу через інтуїтивно зрозумілий графічний інтерфейс користувача (GUI), суворо дотримуючись при цьому найсуворіших сучасних стандартів та патернів веб проектування.

1.1.2. Опис цілей, цільової аудиторії та основних сценаріїв використання

Практична ефективність, комерційна або соціальна успішність будь-якого програмного продукту безпосередньо та нерозривно залежить від того, наскільки точно і повно він відповідає глибинним потребам та очікуванням своєї цільової аудиторії. Науковий аналіз користувацького досвіду (User Experience - UX) на етапі проєктування вимагає детального, скрупульозного сегментування майбутніх користувачів та чіткої формалізації сценаріїв їхньої повсякденної поведінки у віртуальному середовищі системи. Цільова аудиторія вебзастосунку ProjectX є принципово неоднорідною за своїм складом, технічними навичками та мотивацією, і тому об'єктивно поділяється на кілька ключових категорій (або персон). Для кожної з цих категорій інженерами передбачені строго специфічні, оптимізовані користувацькі сценарії взаємодії (Use Cases).

Базові, фундаментальні сценарії використання розроблюваної системи охоплюють наступні рольові моделі та їхні аспекти:

- **Неавторизовані відвідувачі:** Анонімний користувач системи (гість), який ще не пройшов процедуру реєстрації, обмежений базовим функціоналом, проте вільно може переглядати загальнодоступні події та читати публічні новини. Цей сценарій спрямований на залучення нових членів та первинне інформування.
- **Члени громади:** Повноцінний, зареєстрований користувач, який підтвердив свою ідентичність, отримує розширений рівень доступу: він має повну можливість робити фінансову донацію на потреби організації, а також гнучко редагувати власний персональний профіль, оновлюючи контактні дані.
- **Адміністративний персонал:** Користувач із найвищим рівнем доступу (адміністратор) володіє ексклюзивними системними повноваженнями, що дозволяють йому створювати нові або редагувати існуючі події, а також безперешкодно публікувати офіційні новини від імені організації.

Точне визначення та ізоляція цих поведінкових сценаріїв ще на початковому, концептуальному етапі проєктування дозволяє архітекторам та Frontend-розробникам максимально коректно вибудувати ієрархію візуального інтерфейсу, грамотно налаштувати захищену маршрутизацію (routing) у клієнтському додатку та спроєктувати оптимальний, найкоротший шлях користувача (User Journey Map). Усе це в кінцевому підсумку кардинально впливає на загальний рівень психологічного комфорту та задоволеності (Customer Satisfaction) від щоденного використання спроєктованої інформаційної системи.

1.1.3. Функціональні, нефункціональні вимоги та вимоги до доступності

Формалізація, документування та затвердження інженерних вимог є, без перебільшення, фундаментальним та найважливішим етапом класичної інженерії програмного забезпечення. Вона дозволяє встановити жорсткі, недвозначні межі життєвого циклу проєкту, уникнути неконтрольованого розростання функціоналу (scope creep) та сформувати об'єктивні критерії його успішного тестування і завершення. Згідно з прийнятими стандартами, вимоги чітко класифікуються на функціональні (ті, що визначають, що саме технічно повинна робити система), нефункціональні (ті, що диктують, як саме, з якою продуктивністю та надійністю система повинна працювати) та специфічні, соціально орієнтовані вимоги до доступності.

Функціональні вимоги (Business Logic Requirements): Цей клас вимог детально описує конкретні бізнес-можливості, інструменти та сервіси, які в обов'язковому порядку надаються кінцевому користувачеві через взаємодію з графічним інтерфейсом розроблюваного вебзастосунку.

- Архітектура клієнтської частини повинна передбачати наявність інформативної головної сторінки з вичерпною інформацією про історію та місію церкви.

- Необхідно спроєктувати та реалізувати динамічний каталог подій з розширеним функціоналом багатопараметричної фільтрації та текстового пошуку.
- Інтерфейс застосунку обов'язково має містити спеціалізовану форму контактів та обробки повідомлень для забезпечення безперервного зворотного зв'язку між громадою та адміністрацією.
- Вимагається складна технічна реалізація системи донацій, що концептуально включає в себе безпечну інтеграцію з зовнішнім платіжним шлюзом на рівні бекенду.
- Обов'язковою та невід'ємною умовою є реалізація механізмів управління особистими профілями користувачів, а також створення захищеної адміністративної панелі для глобального управління системою.

Нефункціональні вимоги (Продуктивність, Масштабованість та Безпека): Цей життєво важливий клас вимог виступає гарантом того, що розроблена програмна система буде максимально стійкою до відмов, здатною до горизонтального масштабування та абсолютно безпечною в суворих умовах реальної експлуатації в мережі Інтернет.

- З метою радикального підвищення продуктивності та зменшення часу початкового рендерингу (Time to Interactive), необхідно використовувати сучасні патерни ленивого завантаження (lazy loading) цілих сторінок та окремих важких компонентів за допомогою нативних інструментів `React.lazy` та компонента `Suspense`.
- Для економії пропускної здатності мережі вимагається проводити глибоку оптимізацію графічних матеріалів (зображень), пріоритетно використовувати сучасний формат стиснення webp та повсюдно застосовувати атрибути lazy-loading для медіаконтенту.
- Керуючись фундаментальними принципами інформаційної безпеки, суворо заборонено зберігати будь-які секрети (наприклад, API-ключі або паролі) у клієнтському коді; для виконання всіх чутливих та

криптографічних операцій необхідно використовувати виключно ізольований сервер.

- Фронтенд-частина системи повинна надійно захищати всі форми від зловмисних атак типу CSRF (Cross-Site Request Forgery) та обов'язково дублювати валідацію введених клієнтом даних на стороні бекенду для унеможливлення ін'єкцій.

Вимоги до доступності (Accessibility - a11y) та UX: Інклюзивність та доступність цифрового інтерфейсу вже давно перестали бути просто рекомендацією; це сучасний, закріплений у міжнародних документах стандарт вебпроєктування. Він дозволяє комфортно та повноцінно користуватися системою людям з найрізноманітнішими фізичними можливостями, порушеннями зору чи моторики, а також на пристроях з різними діагоналями екранів.

- Кольорова палітра інтерфейсу має забезпечувати необхідний, законодавчо визначений рівень контрасту; розробники зобов'язані використовувати виключно семантичні HTML-теги та превентивно застосовувати спеціальні ARIA-атрибути при необхідності розширення контексту для екранних дикторів (Screen Readers).
- Критично важливим аспектом зручності є коректний, програмно керований фокус-менеджмент (Focus Management) при відкритті та закритті будь-яких модальних вікон або діалогових панелей.
- У всій системі повинна бути імплементована бездоганна підтримка клавіатурної навігації (відмова від виключного використання миші) та інтеграція ARIA-атрибутів на найнижчому рівні базових компонентів.
- З метою психологічного заспокоєння користувача, для візуального відображення стану завантаження асинхронних даних по мережі мають обов'язково використовуватися сучасні UI-патерни, такі як анімовані лоадери (спінери) та плейсхолдери-скелетони (Skeleton Screens).
- Для забезпечення безпомилкового та коректного введення персональних даних у формах реєстрації чи оплати, в інтерфейсі повинні миттєво

відображатися зрозумілі, контекстно залежні валідаційні повідомлення про помилки.

1.2. Огляд предметної області та аналогів

1.2.1. Опис предметної області та бізнес-процесів організації

Предметна область даного кваліфікаційного дослідження лежить у площині цифровізації управлінських, інформаційних та комунікаційних процесів громадських і релігійних організацій. Сучасна парафія чи релігійна спільнота являє собою складний соціальний організм, який функціонує за певними правилами та генерує великі обсяги інформації. Традиційні бізнес-процеси таких організацій історично спиралися на особисту комунікацію, паперовий документообіг та усні оголошення, що в умовах сучасного ритму життя призводить до інформаційного вакууму та зниження рівня залученості членів громади.

Ключові бізнес-процеси, які підлягають цифровій трансформації та автоматизації в рамках розроблюваної системи ProjectX, включають:

- Інформаційне мовлення та просвітництво: Безперервний процес генерації та розповсюдження новин, офіційних заяв та тематичних матеріалів серед членів громади та зацікавлених осіб. Цей процес вимагає наявності зручної адміністративної панелі для модерації контенту.
- Подієвий менеджмент (Event Management): Систематизація розкладу богослужінь, зустрічей, семінарів та інших заходів. Цей процес передбачає створення каталогу подій з можливістю їх фільтрації та пошуку для кінцевого користувача.
- Управління фінансовими надходженнями: Організація прозорого та безпечного механізму збору добровільних пожертв (донацій), що вимагає безшовної інтеграції з надійним платіжним шлюзом на серверній стороні (бекенді).

- Комунікація та зворотний зв'язок: Забезпечення прямих каналів зв'язку між парафіянами та адміністрацією через спеціалізовані контактні форми, а також обробка цих звернень зі зміною їх статусу.

1.2.2. Аналіз існуючих рішень на ринку (аналоги)

Для того щоб спроєктований програмний комплекс був конкурентоспроможним та доцільним, необхідно провести ґрунтовний критичний аналіз існуючих ринкових альтернатив. Дослідження ринку програмного забезпечення для церковних організацій (Church Management Software - ChMS) виявляє кілька поширених підходів до вирішення проблеми їх цифровізації.

1. Використання соціальних мереж та месенджерів (Telegram, Facebook, Instagram): Цей підхід є найпоширенішим завдяки нульовому порогу входження та відсутності фінансових витрат. Проте він має критичні недоліки: алгоритмічні стрічки новин обмежують охоплення аудиторії, інформація швидко губиться в загальному потоці, а специфічний функціонал (наприклад, складні системи донацій чи структуровані календарі подій) неможливо реалізувати в рамках закритої екосистеми соціальної мережі.
2. Універсальні системи управління контентом (CMS) (наприклад, WordPress): Використання готових CMS дозволяє швидко розгорнути вебсайт. Однак такі системи є архітектурно монолітними, що суттєво ускладнює їх оптимізацію та масштабування. Крім того, адаптація універсальної CMS під специфічні потреби (наприклад, кастомні профілі користувачів та управління специфічними даними подій) вимагає встановлення великої кількості сторонніх плагінів, що негативно впливає на продуктивність та створює серйозні вразливості в системі інформаційної безпеки.
3. Спеціалізовані платформи SaaS: Існують потужні хмарні сервіси для церков, які пропонують багатий функціонал "з коробки". Їх головними

недоліками є висока вартість підписки, складна локалізація інтерфейсів для вітчизняного ринку та абсолютна відсутність контролю над власними даними, які зберігаються на чужих серверах.

1.2.3. Висновки щодо необхідного функціоналу

Всебічний та багатогранний аналіз поточного стану ринку інформаційних технологій у контексті цифровізації діяльності громадських та релігійних організацій дозволяє зробити фундаментальний та стратегічно обґрунтований висновок. Існуючі ринкові альтернативи — від універсальних систем управління контентом (CMS) до закритих хмарних платформ (SaaS) — не здатні повною мірою задовольнити унікальний комплекс потреб сучасної громади, що вимагає розробки власного, архітектурно незалежного та функціонально цілісного програмного комплексу типу Full-Stack під робочою назвою ProjectX.

Вибір на користь власної розробки замість використання готових шаблонів зумовлений необхідністю повного суверенітету над даними та гнучкості в імплементації специфічних бізнес-процесів. Формування функціонального ядра системи базується на наступних концептуальних положеннях:

1. **Парадигма розділеної архітектури як фундамент масштабованості** Прийняття рішення про використання розділеної (decoupled) архітектури, де клієнтська та серверна частини функціонують як незалежні одиниці, є єдино правильним та технологічно перспективним шляхом. Це дозволяє уникнути недоліків монолітних систем, де будь-яка зміна в інтерфейсі може призвести до нестабільності серверної логіки. У проєкті ProjectX серверна частина проєктується як високонадійний REST API на базі платформи ASP.NET Core. Такий вибір гарантує не лише колосальну обчислювальну потужність та швидкість обробки транзакцій, а й забезпечує суворе дотримання принципів Clean Architecture та CQRS, що є критично важливим для довгострокового супроводу системи.

2. Стратегічні напрямки серверного функціоналу Серверна інфраструктура бере на себе роль інтелектуального центру, відповідального за:

- **Глибоку ідентифікацію та безпеку:** реалізація складних механізмів автентифікації через JWT-токени та інтеграцію з Google OAuth, що забезпечує безкомпромісний захист персональних даних парафіян.
- **Централізований менеджмент контенту:** динамічне управління потоками новин та календарем церковних заходів (Events), що дозволяє адміністрації оперативно реагувати на зміни в житті громади.
- **Обробку комунікаційних запитів:** автоматизація збору та класифікації повідомлень від користувачів через спеціалізовані контактні форми з можливістю аудиту їхнього статусу.

3. Клієнтська частина: Single Page Application (SPA) як засіб залучення З боку користувача система повинна сприйматися як безшовне, швидке та інтуїтивно зрозуміле середовище. Використання технології Single Page Application (SPA) для фронтенд-частини дозволяє досягти блискавичної взаємодії, що за своєю якістю та швидкістю відгуку наближається до нативних мобільних застосунків. Клієнтська частина ProjectX фокусується на реалізації таких високорівневих інтерфейсів:

- **Інтерактивний каталог та навігація:** розширені можливості фільтрації та пошуку подій, що мінімізують когнітивне навантаження на користувача.
- **Персоналізований досвід:** створення захищених профілів користувачів, де кожен член громади може управляти власною інформацією та візуалізацією свого профілю.
- **Фінансовий інструментарій:** імплементація безпечного та прозорого інтерфейсу для здійснення донацій, що інтегрується з платіжними шлюзами на стороні бекенду, забезпечуючи фінансову стабільність організації.

4. Синергія та практична цінність обраного підходу Комплексний аналіз показує, що обрана Full-Stack стратегія забезпечує ідеальну синергію між

технічною досконалістю та користувацькою зручністю. Такий підхід гарантує абсолютний контроль над кожним бітом інформації, що циркулює в системі, забезпечує можливість практично безмежного горизонтального масштабування при зростанні кількості користувачів та впроваджує рівень безпеки, недоступний для більшості публічних CMS. У підсумку, спроєктований функціонал ProjectX є не просто набором інструментів, а цілісною цифровою платформою, здатною стати надійним фундаментом для розвитку церковної спільноти в умовах сучасного інформаційного простору.

1.3. Інформаційна архітектура та огляд технологій

1.3.1. Карта сайту, структура сторінок та прототипи навігації

Інформаційна архітектура виступає концептуальним фундаментом будь-якого вебзастосунку, визначаючи логіку розташування контенту та шляхи переміщення користувача по системі. Вона повинна бути інтуїтивно зрозумілою та мінімізувати когнітивне навантаження. Згідно з проєктними вимогами, логічна структура сторінок (маршрутів) клієнтської частини включає наступні основні вузли:

- **Головна сторінка (Home):** Презентаційна панель з базовою інформацією про організацію.
- **Каталог (Events):** Сторінка з переліком заходів, яка підтримує складну фільтрацію.
- **Модуль фінансування (Donate):** Спеціалізована сторінка для оформлення пожертв.
- **Особистий кабінет (Profile):** Захищений простір для авторизованих користувачів.
- **Панель керування (Admin):** Ізольована зона для управління контентом та даними системи.

Для забезпечення наскрізної навігації розроблено структурні компоненти обгортки (Layout). Компонент Header виконує функцію головного меню та містить кнопки логіну/реєстрації, елементи навігації та панель пошуку. Нижня частина інтерфейсу інкапсульована в компонент Footer, який агрегує контактну інформацію та посилання на соціальні мережі.

1.3.2. Вибір технологій для Frontend (React, Redux Toolkit, Vite)

Вибір технологічного стеку для клієнтської частини базується на сучасних галузевих стандартах розробки високонавантажених SPA.

- **Бібліотека відображення:** Базовим інструментом для побудови UI було обрано бібліотеку React. Вона дозволяє реалізувати суворий компонентний підхід, де презентаційні (dumb) компоненти відповідають виключно за візуалізацію, а контейнерні (smart) — за логіку та взаємодію зі станом.
- **Управління станом:** Для вирішення проблеми синхронізації глобального стану додатку використовується Redux Toolkit (слайси, функція createAsyncThunk). Це дозволяє централізовано зберігати інформацію про сесію користувача та кешувати асинхронні дані. Крім того, для кешування та нормалізації даних може застосовуватися RTK Query або ручна стратегія.
- **Комунікація з API:** Для виконання HTTP-запитів до бекенду інтегровано клієнт axios з централізованою конфігурацією (api/axiosConfig.js), де налаштовані інтерсептори для автоматичного додавання заголовка Authorization та глобальної обробки помилок (toasts).
- **Валідація та тестування:** Перевірка даних форм на клієнті здійснюється за допомогою бібліотеки Yup у зв'язці з Formik (або React Hook Form). Якість коду гарантується написанням юніт-тестів для компонентів з використанням Jest та React Testing Library.

- **Збірка:** У якості інструменту збірки проєкту (bundler) обрано Vite, що забезпечує миттєве гаряче перезавантаження модулів під час розробки (HMR) та ефективну оптимізацію коду для продакшену.

1.3.3. Вибір технологій для Backend (ASP.NET Core, бази даних)

Серверна інфраструктура є серцем інформаційної системи, що вимагає використання надійних, статично типізованих та продуктивних платформ.

- **Фреймворк:** Для розробки бекенду було обрано платформу .NET 8 (LTS) та мову програмування C#. Вибір ASP.NET Core обґрунтований його надзвичайною швидкістю обробки HTTP-запитів, кросплатформністю та потужним вбудованим механізмом впровадження залежностей (Dependency Injection).
- **Архітектурні патерни:** Проєкт суворо дотримується принципів чистої архітектури (Clean Architecture), розділяючи систему на шари API, Application, Domain та Infrastructure. Для обробки бізнес-логіки успішно імплементовано патерн CQRS (Command Query Responsibility Segregation) за допомогою бібліотеки MediatR.
- **База даних та ORM:** Як основне сховище даних використовується реляційна база даних SQL Server, схема якої спроєктована за принципами 3NF (третьої нормальної форми). Взаємодія з базою даних інкапсульована за допомогою патерну Repository, а безпосередній доступ до даних (об'єктно-реляційне відображення) здійснюється через Entity Framework Core.
- **Безпека:** Для гарантування безпеки впроваджено систему валідації на базі FluentValidation, криптографічне хешування паролів (алгоритм bcrypt) та механізм автентифікації на основі JWT-токенів (Asp.Net.Core.Authentication.JwtBearer), а також інтеграцію з Firebase Admin для підтримки Google OAuth.

РОЗДІЛ 2. АРХІТЕКТУРА ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО КОМПЛЕКСУ

2.1. Архітектура серверної частини та бази даних

Серверна інфраструктура (Backend) виступає концептуальним ядром та "мозковим центром" усієї інформаційної системи. Вона бере на себе найважчі обчислювальні завдання, гарантує консистентність даних, реалізує складну бізнес-логіку та забезпечує захист інформації від несанкціонованого доступу.

2.1.1. Загальна структура бекенду (Clean Architecture, CQRS)

Для бездоганної технічної та методологічної реалізації серверної частини було обрано передовий архітектурний підхід **Clean Architecture** (Чиста архітектура), концептуалізований та популяризований видатним інженером Робертом Мартіном (відомим як Uncle Bob). Ця парадигма є еволюційним розвитком гексагональної архітектури (Ports and Adapters) та цибулевої архітектури (Onion Architecture). Вона передбачає глибоку структурну декомпозицію проєкту на чотири повністю незалежні концентричні шари: Domain, Application, Infrastructure та API. Головний непорушний постулат цього підходу — суворе "правило залежностей" (The Dependency Rule), згідно з яким вектор залежностей на рівні вихідного коду завжди спрямований виключно ззовні всередину. Це робить бізнес-логіку абсолютно ізольованою та незалежною від змін у мінливих веб-фреймворках, механізмах відображення чи специфічних діалектах баз даних.

- **Domain (Доменний шар):** Це найглибший, найчистіший рівень абстракції, що знаходиться в самому центрі архітектури. Він містить базові сутності предметної області (Entities, такі як User, Event, News), об'єкти значень (Value Objects) та фундаментальні, незмінні бізнес-правила організації. Цей шар не має жодного уявлення про те, як дані зберігаються на диску або як вони передаються по мережі.

- **Application (Шар застосунку):** Саме тут інкапсулюється оперативна, специфічна для конкретного додатку бізнес-логіка (Use Cases). На цьому рівні імплементовано потужний архітектурний патерн **CQRS** (Command Query Responsibility Segregation). Теоретичне підґрунтя CQRS полягає у фізичному та логічному розділенні операцій, що модифікують стан системи (Commands), від операцій, що лише зчитують дані без їх зміни (Queries). Такий підхід усуває конфлікти при одночасному доступі до даних та відкриває шлях до асиметричного масштабування. Використання спеціалізованої бібліотеки MediatR забезпечує побудову архітектури, керованої подіями та повідомленнями, де компоненти взаємодіють між собою максимально слабко (loose coupling), а кожен запит обробляється ізольованим обробником (Handler).
- **Infrastructure (Шар інфраструктури):** Цей зовнішній шар бере на себе відповідальність за взаємодію з матеріальним світом технологій. Він реалізує доступ до реляційної бази даних SQL Server через використання потужного ORM-фреймворку Entity Framework Core, керує версіонуванням схеми через міграції та імплементує абстрактні репозиторії (Repository Pattern), контракти (інтерфейси) яких були задекларовані у внутрішньому шарі Application.
- **API (Шар представлення):** Фасад та шлюз системи, представлений ASP.NET Core контролерами. Їхня єдина відповідальність — маршрутизувати вхідні HTTP-запити від гетерогенних клієнтських додатків, валідувати транспортні об'єкти (Data Transfer Objects — DTO), формувати об'єкти команд чи запитів для медіатора та перетворювати отримані результати у стандартизовані JSON-відповіді з коректними HTTP статус-кодами.

2.1.2. Схема реляційної бази даних та взаємодія сутностей

Для забезпечення абсолютної цілісності (Integrity), консистентності (Consistency) та довгострокової надійності зберігання великих масивів структурованої інформації було спроєктовано реляційну базу даних, яка суворо відповідає математичним правилам нормалізації, зокрема, Третій нормальній формі (3NF). Дотримання 3NF унеможливлює виникнення небезпечних аномалій оновлення, видалення чи вставки даних, а також радикально зменшує надлишковість збереженої інформації (Data Redundancy). Вибір Microsoft SQL Server як основної СУБД обґрунтований його повною відповідністю вимогам ACID (Atomicity, Consistency, Isolation, Durability), що є критично важливим для систем, які оперують фінансовими або особистими даними.

Логічна модель даних оперує складним комплексом взаємопов'язаних таблиць. Центральним хабом усієї бази виступає таблиця користувачів (Users), яка має реляційні зв'язки типу "багато-до-одного" з довідником ролей (Roles) та зв'язки типу "один-до-багатьох" з мультимедійним сховищем зображень профілю (UserImages). Для забезпечення найвищого рівня інформаційної безпеки та унеможливлення атак типу Insecure Direct Object Reference (IDOR) чи перебору ідентифікаторів (Enumeration Attacks), замість традиційних автоінкрементних цілих чисел як первинних ключів (Primary Keys) повсюдно використовуються глобально унікальні ідентифікатори (GUID).

Крім того, система містить незалежні, доменно-специфічні модульні таблиці для управління церковними заходами (Events) та новинним контентом (News). Ці таблиці оснащені спеціальними B-Tree індексами для пришвидшення повнотекстового пошуку та екстремальної оптимізації сортування вибірок за датою. Критично важливим інженерним рішенням у контексті глобалізації застосунку є зберігання всіх без винятку часових міток (timestamps) у базі даних виключно у форматі Coordinated Universal Time (UTC).

2.1.3. Модуль автентифікації, JWT-токени та безпека бекенду

У сучасних умовах перманентного зростання кількості та складності кіберзагроз, безпека програмного забезпечення є беззаперечним пріоритетом нульового рівня. Підсистема безпеки проєкту розроблена на основі парадигми "Глибокого ешелонованого захисту" (Defense in Depth), яка передбачає наявність кількох незалежних рівнів контролю. Під час процедури реєстрації нових користувачів, їхні паролі проходять обов'язкову процедуру незворотного криптографічного хешування за допомогою ресурсоємного та криптостійкого алгоритму bcrypt. Цей алгоритм, на відміну від застарілих MD5 чи SHA-256, використовує динамічне "соління" (salting) та має настроюваний фактор складності (work factor), що робить технічно неможливим і фінансово недоцільним відновлення оригінального пароля за допомогою словникових атак (Dictionary Attacks) чи використання райдужних таблиць (Rainbow Tables) навіть у гіпотетичному випадку повного дампу бази даних зловмисниками.

Авторизація доступу до захищених ресурсів реалізована за сучасним протоколом без збереження стану (Stateless Authentication) з використанням відкритого індустріального стандарту JWT (JSON Web Tokens). Цей підхід кардинально відрізняється від класичних сесій: після успішної ідентифікації сервер генерує токен, що містить зашифровані (або підписані) дані (payload) про користувача, і передає його клієнту. Клієнтський додаток зобов'язаний додавати цей токен до HTTP-заголовка `Authorization: Bearer` при кожному наступному мережевому зверненні до захищених ендпоінтів. Такий механізм позбавляє сервер необхідності зберігати стан сесії в оперативній пам'яті або базі даних, що є критичною передумовою для безперешкодного горизонтального масштабування бекенду.

Окрім класичної локальної автентифікації, проєкт глибоко інтегровано з потужною екосистемою Google OAuth через використання серверного SDK Firebase Admin. Це дозволяє імплементувати механізм Single Sign-On (SSO), завдяки якому користувачі можуть миттєво та максимально безпечно входити в систему, делегуючи складну процедуру перевірки ідентичності високозахищеним серверам корпорації

Google. Контроль доступу на рівні HTTP-контролерів жорстко та імперативно регламентується суворою рольовою моделлю доступу (Role-Based Access Control — RBAC), надійно розмежовуючи повноваження, системні функції та видимість даних між ролями Admin, User та Moderator.

2.2. Архітектура клієнтської частини (Frontend)

Фронтенд-частина розроблюваного вебзастосунку спроєктована у вигляді високопродуктивного, сучасного односторінкового додатка (Single Page Application — SPA). На відміну від класичних багатосторінкових сайтів (Multi-Page Applications), де кожна дія користувача призводить до важкого перезавантаження всієї сторінки з сервера, SPA завантажує базовий HTML, CSS та JavaScript-код лише один раз при першому візиті. Після цього всі подальші оновлення інтерфейсу та маршрутизація відбуваються виключно на стороні клієнта за рахунок динамічного маніпулювання DOM-деревом. Це створює для користувача неперевершене відчуття швидкості, плавності та безперервності взаємодії, яке раніше було характерне виключно для нативних мобільних застосунків.

2.2.1. Структура проєкту, компонентний підхід та управління станом

Клієнтський застосунок спирається на використання екосистеми **React** — найсучаснішої та найпопулярнішої у світі декларативної JavaScript-бібліотеки для побудови користувацьких інтерфейсів, яка революціонізувала веброзробку завдяки інноваційній концепції Virtual DOM та алгоритму узгодження (Reconciliation). Структура кодової бази проєкту суворо ієрархізована та логічно сегрегована на модулі (components/, pages/, features/, api/, utils/). У розробці імперативно дотримується філософія компонентно-орієнтованого проєктування. Вона полягає в декомпозиції складного, монолітного інтерфейсу на набір дрібних, абсолютно ізольованих, інкапсульованих та придатних для багаторазового використання частин. При цьому компоненти чітко та концептуально поділяються на презентаційні (dumb components), які нічого не знають про бізнес-логіку і відповідають виключно за рендеринг UI на основі вхідних пропсів, та контейнерні (smart components), що підключені до глобального сховища стану, оркеструють дані та обробляють асинхронні побічні ефекти.

Оскільки SPA-додатки за своєю природою генерують та обробляють колосальну кількість динамічних даних у пам'яті клієнта (статус авторизації, отримані з бекенду великі масиви списків подій, вміст багатовимірних модальних вікон, стан заповнення форм), виникає проблема складності передачі даних між компонентами (так званий prop-drilling). Для ефективного, детермінованого та передбачуваного глобального управління станом (Global State Management) інтегровано еталонну архітектурну бібліотеку Redux Toolkit. Її використання (через сучасні механізми слайсів та генераторів асинхронних екшенів createAsyncThunk) забезпечує беззаперечне дотримання фундаментального принципу єдиного джерела істини (Single Source of Truth) та концепції однонаправленого потоку даних (Unidirectional Data Flow). Завдяки цьому стан додатка стає імутабельним (immutable), його зміни можна відстежувати у часі (Time-Travel Debugging), що робить поведінку інтерфейсу максимально передбачуваною та на порядки спрощує відлагодження програми розробниками.

2.2.2. Візуальний дизайн, палітра кольорів та адаптивність мобільного дизайну

Естетична привабливість, візуальна гармонія та інтуїтивна ергономічність (Usability) інтерфейсу є не просто допоміжними атрибутами, а визначальними факторами комерційного успіху та рівня утримання користувачів (User Retention) будь-якого цифрового продукту. Візуальний дизайн застосунку ProjectX розроблено на стику науки про когнітивне навантаження та сучасних трендів UI-дизайну. Він відзначається стилем розумного мінімалізму (Flat Design), забезпеченням оптимального рівня контрасту для покращення читабельності (відповідно до стандартів WCAG) та використанням фірмової, ретельно підібраної палітри кольорів, що психологічно резонує з цінностями релігійної та громадської організації.

Фундаментом побудови інтерфейсу є власна, кастомізована бібліотека уніфікованих UI-компонентів (Button, Input, Card, Modal, Toast), які спроєктовані

за принципами Design Systems. Вони мають суворо стандартизовані властивості та API (variant, size, disabled, onClick) і гарантують абсолютну візуальну цілісність, консистентність та впізнаваність системи на всіх її екранах. З огляду на безперечний глобальний тренд експоненційного зростання мобільного трафіку, розробка велася за методологією Mobile-First. Інтерфейс є абсолютно адаптивним (Responsive Web Design): за допомогою сучасних CSS-фреймворків та медіа-запитів він коректно реагує на зміну роздільної здатності екрана чи орієнтації пристрою, плавно трансформуючи сітки (Grid/Flexbox) та масштабуючи розміри шрифтів для забезпечення ідеального відображення як на широкоформатних десктопних моніторах (4K), так і на компактних екранах мобільних пристроїв.

2.2.3. Інтеграція з Backend: налаштування HTTP-клієнта (Axios), перехоплювачі та обробка помилок

Навіть найдосконаліша клієнтська частина була б абсолютно статичною та марною без надійного, безперебійного та безпечного механізму обміну даними з серверною інфраструктурою. Для реалізації складного мережевого шару комунікації замість нативного API fetch використовується потужний, багатофункціональний HTTP-клієнт Axios, який глобально конфігурований у спеціалізованому модулі (api/axiosConfig.js). Фундаментальним архітектурним рішенням у цьому аспекті, що базується на принципах аспектно-орієнтованого програмування (AOP), є використання механізму перехоплювачів (interceptors).

Перехоплювачі запитів (Request Interceptors) діють як своєрідне клієнтське middleware. Вони автоматично ін'єктують JWT-токен у заголовок Authorization перед відправкою кожного захищеного HTTP-запиту на сервер, позбавляючи розробників необхідності ручного дублювання цього коду в кожному окремому компоненті або сервісі. Своєю чергою, перехоплювачі відповідей (Response Interceptors) забезпечують централізовану, глобальну та уніфіковану обробку мережевих помилок. Наприклад, у разі отримання від бекенду статусу 401 Unauthorized (означає, що вичерпано термін дії токена, ExpirationMinutes) або 400 Bad Request (серверна

помилка валідації DTO), система автоматично перериває виконання поточної операції, очищує стан сесії (якщо потрібно) та виводить на екран користувача елегантне, неблокуюче спливаюче сповіщення (Toast) з детальним, локалізованим поясненням причини збою. Це кардинально покращує користувацький досвід (UX), оскільки користувач завжди розуміє стан системи.

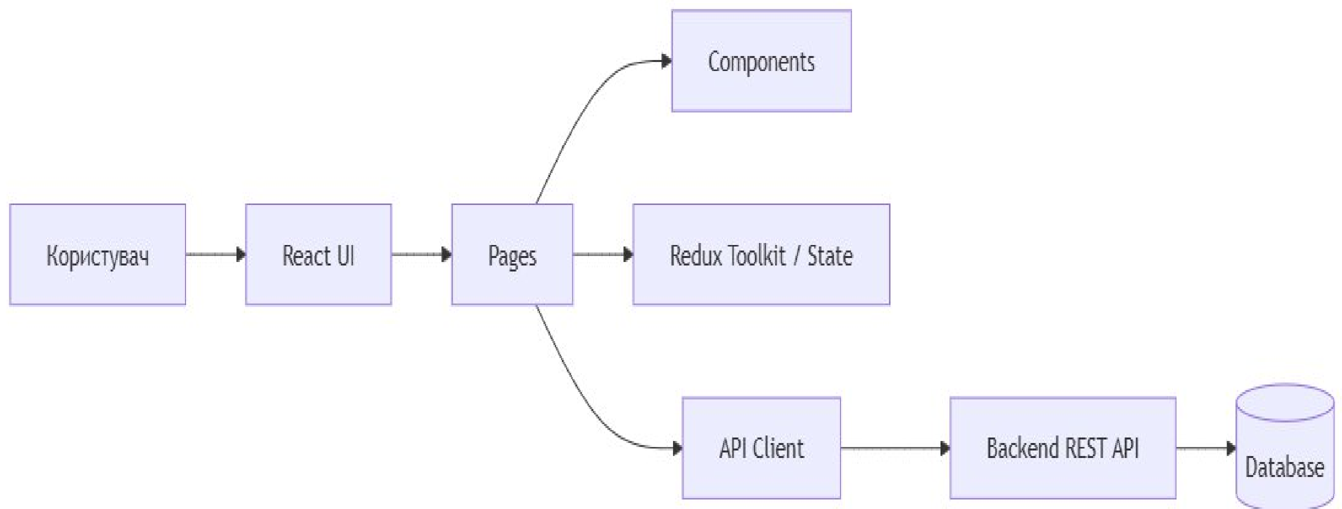

Рис.2.2.3.Діаграма архітектури клієнта / Приклад запиту API.

2.3. Реалізація функціональних модулів застосунку

Згідно з методологією модульного проєктування та поділу на домени, весь багатий функціонал застосунку логічно та фізично розбитий на самостійні, слабко пов'язані між собою бізнес-модулі. Це дозволяє командам розробників працювати над різними частинами додатку паралельно без ризику виникнення конфліктів злиття коду.

2.3.1. Каталог подій, новин та фільтрація

Модуль відображення подій та новин виступає основною інформаційною вітриною та "обличчям" організації у цифровому просторі. Він реалізує глибоку, асинхронну інтеграцію з відповідними REST API ендпоінтів бекенду (наприклад,

GET /api/events та GET /api/news) за допомогою HTTP GET запитів. Зважаючи на потенційно величезний обсяг накопичених архівних даних, як на рівні бази даних сервера, так і на рівні клієнтського рендерингу імплементовано ефективні алгоритми посторінкової пагінації (Pagination) за параметрами page та pageSize. Це архітектурне рішення дозволяє завантажувати інформацію суворо дозованими порціями, мінімізуючи навантаження на пропускну здатність мережі, економлячи батарею мобільного пристрою та зменшуючи споживання оперативної пам'яті браузера. Користувач отримує доступ до розширених, багатокритеріальних інструментів фільтрації, які дозволяють миттєво знаходити потрібні церковні заходи чи статті за заданими критеріями. Більше того, під час неминучого очікування відповіді від сервера (Network Latency), інтерфейс тимчасово відображає ергономічні анімовані плейсхолдери (скелетони, Skeleton Screens) для уникнення ефекту психологічного "завмирання" екрану та забезпечення ефекту високої уявної продуктивності (Perceived Performance).

2.3.2. Система реєстрації, профілі користувачів та адміністративна панель

Модуль керування ідентичністю (Identity Management) охоплює та супроводжує повний життєвий цикл перебування користувача в системі. Складний процес реєстрації (Onboarding) супроводжується миттєвою, багатофакторною валідацією даних безпосередньо на стороні клієнта за допомогою спеціалізованих абстрактних бібліотек (наприклад, Formik у зв'язці зі схемним валідатором Yup). Таке превентивне проєктування дозволяє виявляти друкарські помилки або невідповідності форматам (наприклад, невірний формат електронної пошти чи занадто слабкий пароль) ще до моменту формування та відправки дорогого HTTP-запиту на сервер, економлячи ресурси бекенду. Після успішної криптографічної авторизації користувачеві відкривається доступ до закритого, персоналізованого особистого профілю. Там він отримує можливості управління своїм цифровим представництвом: може редагувати свої персональні дані через PUT /api/users/{id} та керувати мультимедійними

файлами, зокрема завантажувати високоякісну фотографію профілю через спеціальний multipart/form-data ендпоінт POST /api/users/{id}/upload-image. Для спеціального персоналу з розширеними правами доступу (Admin) передбачена ізольована, захищена від звичайних відвідувачів адміністративна панель (Dashboard), де вони можуть здійснювати всі види CRUD-операцій (створення, читання, оновлення, видалення) над системним контентом, публікувати новини, а також гнучко управляти ролями інших користувачів через PUT /api/users/{userId}/roles.

2.3.3. Система донацій, платіжний інтерфейс та контактні форми

Фінансова життєздатність, стабільність діяльності організації та ефективний зворотний зв'язок реалізуються через цей критично важливий, високочутливий модуль. Система донацій (пожертв) розроблена з неухильним урахуванням найвищих світових стандартів транзакційної безпеки та ідемпотентності мережевих запитів. Клієнтський додаток генерує криптографічно захищений запит на відповідний ендпоінт бекенду (POST /donations), ініціюючи складний процес міжсерверної інтеграції з зовнішнім сертифікованим платіжним шлюзом (Payment Gateway). Окрім складних фінансових аспектів, цифрова екосистема пропонує користувачам можливість прямої, структурованої комунікації через спеціальні контактні форми. Текстові повідомлення асинхронно відправляються через метод POST /api/user-messages і надійно акумулюються в базі даних бекенду, де адміністратори можуть ефективно проводити їх аудит, класифікувати та управляти їхніми статусами (IsRead маркер через PUT запит). Це гарантує створення надійного каналу комунікації та забезпечує гарантовану, своєчасну обробку кожного звернення або запиту від парафіян.

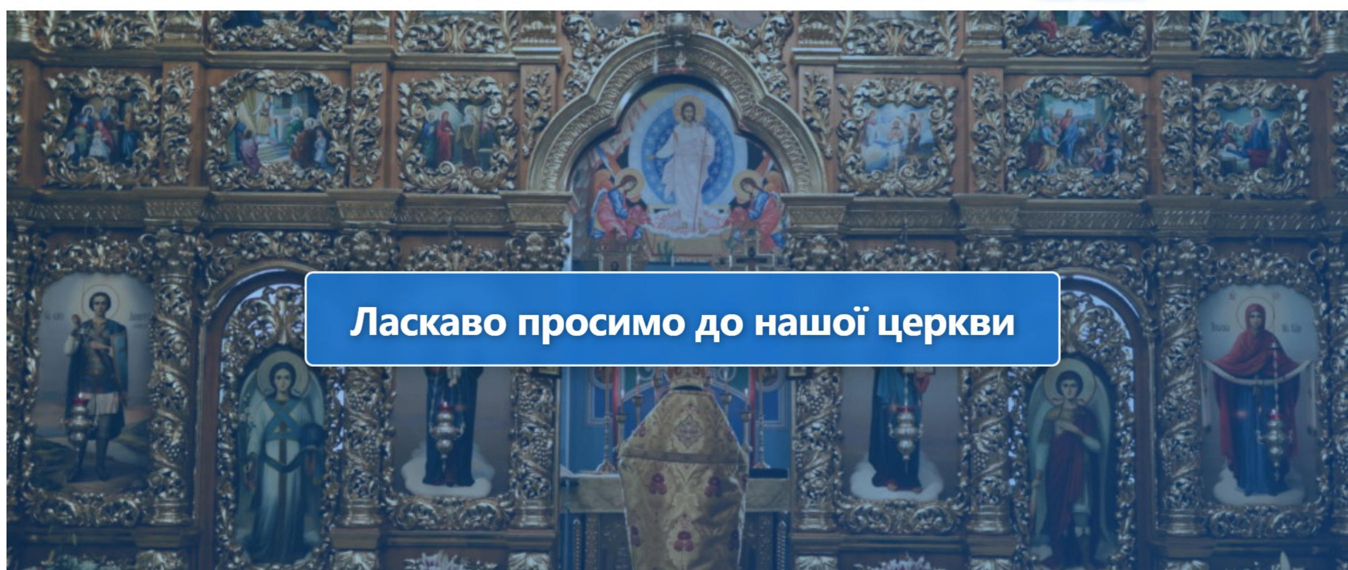

Рис.2.3.3.Макет головної сторінки

РОЗДІЛ 3. ТЕСТУВАННЯ, РОЗГОРТАННЯ ТА СУПРОВІД

Сучасна парадигма розробки програмного забезпечення беззаперечно диктує, що процес створення інформаційної системи не завершується на етапі написання вихідного коду та проєктування структур баз даних. Завершальна фаза життєвого циклу розробки (Software Development Life Cycle — SDLC), яка охоплює всебічну верифікацію та валідацію якості, автоматизацію конвеєрів розгортання та стратегічне планування подальшого супроводу, є критично важливою, а часто і найскладнішою складовою для забезпечення довгострокової життєздатності продукту в реальних умовах експлуатації. Практика доводить, що вартість виправлення програмного дефекту зростає експоненціально залежно від того, на якому етапі він був виявлений: помилка, знайдена на етапі тестування, коштує в десятки разів дешевше для бізнесу чи організації, ніж критичний збій, виявлений користувачами у продуктивному середовищі (Production). У даному розділі детально та ґрунтовно розглянуто методологію комплексного тестування Full-Stack застосунку ProjectX, архітектуру сучасних конвеєрів безперервної інтеграції та доставки (CI/CD), а також стратегії довгострокового супроводу, резервного копіювання та масштабування системи.

3.1. Організація тестування та якість коду

Тестування програмного забезпечення — це фундаментальна інженерна та наукова дисципліна, що спрямована на емпіричне виявлення дефектів (багів), сувору перевірку відповідності реалізованого функціоналу початковим технічним специфікаціям та гарантування стабільності системи під час штатного чи пікового навантаження. Для забезпечення найвищого рівня якості проєкту ProjectX була обрана превентивна стратегія багаторівневого тестування (Shift-Left Testing Paradigm). Ця стратегія базується на класичній моделі "Піраміди тестування" Майка Кона, де масивну основу складають швидкі, ізольовані одиничні тести (Unit Tests), середній рівень формують інтеграційні перевірки (Integration Tests), а вершину —

комплексні наскрізні тести (End-to-End Tests) та перевірки на стороні користувацького інтерфейсу.

3.1.1. Юніт-тестування для ізольованих UI-компонентів та бекенд-сервісів

Одиничне (або модульне) тестування є першим, фундаментальним етапом верифікації вихідного коду, що дозволяє перевірити працездатність найменших логічних модулів системи (класів, методів, функцій, компонентів) у стані абсолютної ізоляції від будь-яких зовнішніх залежностей (таких як файлова система, мережа чи база даних). Це досягається завдяки широкому застосуванню принципів інверсії управління (Inversion of Control) та впровадження залежностей (Dependency Injection).

- **Бекенд-сервіси (.NET 8):** На серверній частині юніт-тести суворо зосереджені на перевірці чистої бізнес-логіки, яка інкапсульована в шарі Application. Для реалізації цих перевірок використовується потужний тестовий фреймворк xUnit у комбінації з бібліотекою створення імітацій (mock-об'єктів) Moq. Використання Moq дозволяє створювати ізольовані тестові сценарії для обробників команд MediatR (Command/Query Handlers), де реальні екземпляри інфраструктурних репозиторіїв програмно замінюються заглушками. Це гарантує, що складна логіка реєстрації нового користувача чи алгоритм створення церковної події працюють абсолютно коректно, незалежно від поточного стану реляційної бази даних SQL Server чи доступності мережі. Також ретельному одиничному тестуванню підлягають класи валідації, побудовані на базі бібліотеки FluentValidation, що забезпечують строгу синтаксичну та логічну перевірку вхідних Data Transfer Objects (DTO) на наявність порожніх полів чи перевищення лімітів символів.
- **UI-компоненти (React):** На стороні клієнтського застосунку юніт-тестування реалізується за допомогою індустріального стандарту — фреймворку Jest у синергії з бібліотекою React Testing Library. На відміну від застарілих підходів, які тестували внутрішній стан компонентів, React

Testing Library фокусується на тестуванні поведінки компонента з точки зору кінцевого користувача. Основна увага приділяється презентаційним компонентам (dumb components) для перевірки коректності рендерингу візуального інтерфейсу при передачі різноманітних комбінацій пропсів (props). Ізольовано тестуються стани кнопок (активна/неактивна), інпутів, карток подій, а також перевіряється коректність виклику функцій зворотного зв'язку (callbacks) при імітації взаємодії користувача (натискання, введення тексту).

3.1.2. Інтеграційні тести для перевірки взаємодії клієнта з API

Якщо одиничні тести гарантують, що кожен окремий гвинтик системи працює правильно, то інтеграційне тестування спрямоване на перевірку коректності зв’язків та інтерфейсів між цими гвинтиками — тобто між окремими шарами архітектури та зовнішніми сервісами. Це є критично важливим етапом для складних Full-Stack проєктів, де навіть незначна помилка у форматі JSON-відповіді або неправильний HTTP статус-код може призвести до ланцюгового збою всього клієнтського застосунку.

- **Серверні інтеграційні тести:** Для серверної архітектури проєкт включає спеціалізовану, фізично відокремлену директорію Api.Tests.Integration, де розміщуються тести, що комплексно імітують реальні HTTP-запити до контролерів. У цих тестах перевіряється повний, наскрізний шлях проходження запиту: від отримання сервером, проходження через Middleware (зокрема, механізми авторизації та JWT-верифікації), маршрутизацію (Routing) до безпосередньої взаємодії з тестовою базою даних (in-memory або ізольованим SQL екземпляром) та остаточного повернення очікуваного статус-коду (наприклад, 201 Created при успішному створенні новини або 401 Unauthorized при відсутності токена).

- **Взаємодія Frontend-Backend:** На клієнтській стороні інтеграційне тестування перевіряє правильність формування та виконання викликів API через глобально налаштований HTTP-клієнт Axios. Глибоко тестуються складні патерни асинхронних викликів, імплементовані у вигляді createAsyncThunk (в екосистемі Redux Toolkit). Перевіряється коректність обробки станів життєвого циклу запиту (pending, fulfilled, rejected) у відповідних слайсах (slices) при отриманні реальних даних від mock-сервера (наприклад, за допомогою MSW - Mock Service Worker) або спеціалізованого тестового бекенду. Також перевіряється робота Axios-інтерсепторів, зокрема здатність системи автоматично додавати Bearer-токен до заголовків та коректно реагувати на глобальні помилки мережі.

3.1.3. Аналіз результатів: виявлені помилки, кросбраузерність та перевірка доступності

Завершення фази автоматизованих та мануальних тестів супроводжується глибоким ретроспективним аналізом отриманих метрик якості. Цей аналітичний процес дозволяє виявити архітектурні "вузькі місця" та гарантувати, що система готова до передачі кінцевим споживачам.

- **Аналіз та класифікація дефектів:** Усі виявлені програмні дефекти (баги) суворо класифікуються за рівнем їхньої критичності (Severity) та пріоритетності виправлення (Priority). Завдяки впровадженню глобального обробника помилок на бекенді, що реалізує стандартний інтерфейс IProblemDetailsFactory для форматування JSON-відповідей, а також централізованій системі повідомлень (вспливаючих toasts) на фронтенді, процес локалізації, ідентифікації та усунення багів значно прискорюється. Це забезпечує високий рівень спостережуваності (Observability) системи під час тестування.
- **Кросбраузерність (Cross-Browser Compatibility):** Враховуючи гетерогенність клієнтського середовища, проводиться системне візуальне

та функціональне тестування в різних сучасних браузерах (Google Chrome на базі рушія Blink, Mozilla Firefox на базі Gecko, Apple Safari на базі WebKit). Це дозволяє переконатися у відсутності специфічних для конкретного браузера багів рендерингу та гарантує, що адаптивний дизайн коректно відпрацьовує медіа-запити на будь-яких пристроях.

- **Перевірка доступності (Accessibility - a11y):** Особлива, соціально відповідальна увага приділяється суворим вимогам доступності, які регламентуються міжнародними стандартами (наприклад, WCAG). Проводиться аудит інтерфейсу: перевіряється достатня контрастність кольорів текстових елементів відносно фону, наявність виключно семантичних HTML-тегів, коректний фокус-менеджмент при взаємодії з модальними вікнами та правильна робота ARIA-атрибутів для забезпечення можливості використання системи людьми з вадами зору за допомогою програм-екранних дикторів (Screen Readers).

№	Сценарій	Очікуваний результат	Фактичний результат	Статус
1	Відкрити головну сторінку	Сторінка завантажується без помилок	Сторінка відкривається коректно	Passed
2	Перейти до каталогу подій	Відображається список подій	Список подій завантажено	Passed
3	Надіслати повідомлення через форму контакту	Повідомлення відправляється успішно	Форма проходить валідацію і відправляється	Passed
4	Створити донацію	Відкривається форма донації	Форма працює коректно	Passed
5	Увійти в профіль користувача	Відображається профіль користувача	Профіль відкривається після авторизації	Passed
6	Відкрити адмінпанель	Доступ лише для адміністратора	Доступ обмежено приватним маршрутом	Passed

Рис.3.1.3.Приклад звіту тестування

3.2. Розгортання та CI/CD

У сучасному швидкоплинному світі розробки програмного забезпечення класичний підхід ручного перенесення файлів на сервер (наприклад, через FTP) вважається абсолютно неприйнятним антипатерном (Anti-Pattern). Автоматизація процесів збірки, тестування та доставки коду через методології безперервної інтеграції та безперервного розгортання (Continuous Integration / Continuous Deployment — CI/CD) та філософію DevOps дозволяє повністю нівелювати вплив людського фактору (Human Error), забезпечити відтворюваність середовищ та гарантувати швидку, безпечну доставку нових функцій до кінцевого споживача.

3.2.1. Налаштування збірки клієнта (Vite) та контейнеризація сервера (Docker)

Для підготовки вихідного коду проєкту до суворих умов продуктивного (продакшн) середовища використовуються найсучасніші інструменти пакування, транспіляції та оптимізації ресурсів.

- **Frontend збірка (Bundling):** Клієнтська частина, написана з використанням сучасних стандартів ECMAScript та JSX, збирається за допомогою надшвидкого інструменту Vite (на базі esbuild). Процес збірки, що ініціюється командою npm run build, виконує цілий комплекс трансформацій: мініфікацію та обфускацію JavaScript-коду, оптимізацію CSS-стилів, видалення невикористаного коду (Tree Shaking) та логічне розділення коду на чанки (Code Splitting) для забезпечення максимально швидкого початкового завантаження сторінок та оптимізації показників Core Web Vitals.
- **Контейнеризація Backend (Dockerization):** Серверна частина платформи повністю ізолюється та упаковується в стандартизовані Docker-контейнери. Для цього використовується багатоетапна збірка (Multi-Stage Build) у Dockerfile, де етап компіляції відбувається на базі

важкого образу mcr.microsoft.com/dotnet/sdk:8.0, а фінальний артефакт переноситься у надзвичайно легкий оптимізований образ mcr.microsoft.com/dotnet/aspnet:8.0. Контейнеризація втілює концепцію незмінної інфраструктури (Immutable Infrastructure), що гарантує абсолютну ідентичність поведінки коду на локальному комп'ютері розробника, тестовому сервері та кінцевому продуктивному середовищі, назавжди усуваючи класичну проблему "на моїй машині все працювало".

3.2.2. Створення CI-пайплайну для автоматизованих тестів та деплою

Методологія безперервної інтеграції в проєкті реалізована через створення декларативних пайплайнів (конвеєрів) за допомогою інструментарію GitHub Actions. Цей конвеєр, конфігурація якого зберігається у файлі .github/workflows/deploy.yml, автоматично та детерміновано запускається при кожному новому push-запиті в головну гілку репозиторію (main) та включає наступні суворо послідовні кроки:

1. Клонування вихідного коду (Checkout).
2. Налаштування середовища (Setup .NET) та відновлення залежностей проєкту за допомогою dotnet restore.
3. Компіляція (Build) серверного проєкту без відновлення залежностей для пришвидшення процесу.
4. Обов'язковий запуск набору автоматизованих тестів (dotnet test), що гарантує якість коду та недопущення регресій.
5. Публікація скомпільованого артефакту (dotnet publish) та безпосереднє розгортання в хмарному середовищі Azure за допомогою екшену azure/webapps-deploy@v2 із використанням секретних профілів публікації.

3.2.3. Хмарне розгортання: Vercel для Frontend та Azure для Backend

Зважаючи на вимоги щодо відмовостійкості, безпеки та масштабованості, система ProjectX розгорнута в хмарних інфраструктурах (Cloud Computing Paradigm), що позбавляє необхідності підтримки власних фізичних серверів (On-Premise).

- **Фронтенд**: Деплой клієнтської частини здійснюється на платформу Vercel — лідера у сфері Edge-хостингу. Vercel забезпечує безшовне автоматичне розгортання безпосередньо з гілки main репозиторію GitHub, автоматичну генерацію та управління SSL-сертифікатами від Let's Encrypt, а також доставку статичного контенту через глобальну мережу доставки контенту (CDN), що мінімізує затримки (latency) для користувачів з різних куточків світу.
- **Бекенд**: Серверна частина API та реляційна база даних розгорнуті в екосистемі Microsoft Azure (як сервіси Azure App Service та Azure SQL Database відповідно). Таке інфраструктурне рішення дозволяє гнучко використовувати інструменти глибокого моніторингу Application Insights для збору телеметрії та відстеження технічного стану системи в режимі реального часу, а також легко налаштовувати правила автоматичного горизонтального масштабування (Auto-scaling) при збільшенні навантаження на систему.

3.3. Супровід, документація та перспективи

Реліз програмного продукту та запуск його в комерційну чи громадську експлуатацію — це лише перший крок на його життєвому шляху. Якісний технічний супровід, ведення актуальної документації та чітке стратегічне планування майбутнього розвитку є абсолютними запоруками успіху та довголіття інформаційної системи.

3.3.1. Інтерактивна документація API (Swagger) та інструкції для адміністраторів

У сучасній архітектурі, де клієнт і сервер жорстко розділені, наявність актуальної документації API є життєво необхідною для роботи фронтенд-розробників та можливої майбутньої інтеграції зі сторонніми сервісами.

- **Swagger/OpenAPI:** Серверна частина проєкту побудована таким чином, що автоматично генерує живу, інтерактивну документацію за стандартом OpenAPI (більш відому як Swagger UI). Цей інструмент не лише детально описує всі доступні HTTP ендпоінти, необхідні моделі даних (Request/Response Bodies) та очікувані статус-коди відповідей, але й дозволяє розробникам виконувати тестові запити безпосередньо з інтерфейсу браузера. Це вирішує одвічну проблему розсинхронізації написаного коду та паперової документації.
- **Інструкції користувача:** Для адміністративного персоналу організації підготовлені спеціалізовані текстові посібники та інструкції з ефективного управління контентом новин, створення подій та обробки звернень через зручну адмін-панель.

AboutUs

GET /about-us/get-all

POST /about-us/upload-images

DELETE /about-us/delete-image/{aboutUsId}

Account

POST /account/signup

POST /account/signin

Auth

POST /api/auth/google-login

POST /api/auth/firebase-login

Event

GET /event/get-all

GET /event/get-by-id/{eventId}

POST /event/create

PUT /event/update/{eventId}

DELETE /event/delete/{eventId}

GET /event/search-by-title

Рис.3.3.1.Приклад сторінки документації Swagger

3.3.2. План резервного копіювання даних та відновлення системи

Дані є найціннішим активом будь-якої організації. Безпека персональних даних парафіян, історії пожертв (донацій) та інформаційних публікацій забезпечується комплексними стратегіями аварійного відновлення (Disaster Recovery). На рівні хмарного провайдера Azure для сервісу Azure SQL Database жорстко налаштовано процес автоматичного створення резервних копій (бекапів) з високим рівнем географічної надлишковості (Geo-Redundant Storage) та утриманням даних до 30 днів. Ця стратегія гарантує показники Recovery Point Objective (RPO) та Recovery Time Objective (RTO) на такому рівні, що дозволяє оперативно відновити базу даних на певний момент часу навіть у разі критичних катаклізмів або збоїв у конкретному дата-центрі Microsoft.

3.3.3. Практична цінність, плани подальшого масштабування та розвитку функціоналу

Проєкт ProjectX несе в собі значну соціальну та практичну цінність для релігійних та громадських організацій, пропонуючи їм сучасний, інкапсульований інструмент для ефективної цифровізації процесів масового інформування, планування заходів та централізованого збору добровільних пожертв. Система спроєктована з урахуванням концепції Open-Closed Principle (відкрита для розширення, закрита для модифікації), що дозволяє легко додавати новий функціонал.

Стратегічні плани подальшого розвитку та масштабування включають:

- Впровадження складного модуля поглибленої фінансової звітності для бухгалтерії організації та створення системи управління графіком волонтерів.
- Розробку нативних мобільних застосунків для платформ iOS та Android (наприклад, з використанням фреймворків Flutter або React Native), які будуть безпосередньо споживати існуючий та перевірений RESTful API.

ВИСНОВОК

Успішне виконання кваліфікаційної роботи забезпечило реалізацію поставленої мети — комплексного проєктування, розробки та впровадження надійного, масштабованого та безпечного серверного веб-інтерфейсу (API) для потреб церковної організації. Ця система покликана вирішити проблему розрізненості каналів комунікації та підвищити ефективність роботи організації через сучасні цифрові технології.

В результаті дослідження було обґрунтовано вибір технологічного стеку, віддавши перевагу високопродуктивній та кросплатформній платформі ASP.NET Core (.NET 8) з мовою C# як оптимальному рішенню для створення кастомного RESTful API. Ключовим інженерним досягненням стала реалізація архітектури з суворим дотриманням принципів Clean Architecture та патерну CQRS (Command Query Responsibility Segregation) за допомогою бібліотеки MediatR. Це гарантувало чітке розділення відповідальності, високу згуртованість модулів, мінімальну зв'язність коду та готовність системи до подальшого масштабування.

Розроблений API повністю забезпечує функціональні вимоги до управління інформаційним контентом та взаємодією:

- Реалізовано надійний модуль автентифікації та авторизації з використанням безпечного хешування паролів (bcrypt), генерацією JWT-токенів та можливістю безшовної інтеграції через Google OAuth.
- Впроваджено повноцінну RESTful архітектуру для управління новинами та подіями, включаючи підтримку пагінації та зберігання часових міток у форматі UTC для уникнення часових аномалій.
- Створено захищений канал зворотного зв'язку (модуль UserMessages), доступ до якого суворо контролюється за допомогою рольової моделі RBAC (Admin, User, Moderator).

Якість, стабільність та відповідність системи початковим вимогам підтверджена багаторівневим тестуванням, яке включало одиничні та комплексні інтеграційні тести. Проєкт повністю підготовлений до промислового розгортання,

використовуючи технологію контейнеризації Docker та оптимізований для роботи у хмарному середовищі Microsoft Azure.

Практична цінність роботи полягає у створенні готового до експлуатації, повністю контрольованого та надійно захищеного інформаційного фундаменту. Це дозволяє церковній організації оптимізувати адміністративні процеси, автоматизувати інформування аудиторії та запропонувати сучасний, високопродуктивний сервіс для цифрової взаємодії з прихожанами.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft Learn. Clean Architecture. — URL: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures (дата звернення: 10.05.2026).
2. Microsoft Learn. CQRS pattern. — URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs (дата звернення: 10.05.2026).
3. React Documentation. — URL: https://react.dev/ (дата звернення: 10.05.2026).
4. Redux Toolkit Official Documentation. — URL: https://redux-toolkit.js.org/ (дата звернення: 10.05.2026).
5. Vite Documentation. — URL: https://vitejs.dev/ (дата звернення: 10.05.2026).
6. React Bootstrap Documentation. — URL: https://react-bootstrap.github.io/ (дата звернення: 10.05.2026).
7. Entity Framework Core Documentation. — URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 10.05.2026).
8. ASP.NET Core Documentation. — URL: https://learn.microsoft.com/en-us/aspnet/core/ (дата звернення: 10.05.2026).
9. MediatR Documentation. — URL: https://github.com/jbogard/MediatR (дата звернення: 10.05.2026).
10. FluentValidation Documentation. — URL: https://fluentvalidation.net/ (дата звернення: 10.05.2026).
11. Fowler M. CQRS. — URL: https://martinfowler.com/bliki/CQRS.html (дата звернення: 09.05.2026).
12. JSON Web Tokens (JWT). — URL: https://jwt.io/ (дата звернення: 10.05.2026).
13. Firebase Authentication Documentation. — URL: https://firebase.google.com/docs/auth (дата звернення: 10.05.2026).

14. OWASP Top 10:2021 — The Ten Most Critical Security Risks to Web Applications.
15. Docker Documentation. — URL: https://docs.docker.com/ (дата звернення: 10.05.2026).
16. Azure Architecture Center. — URL: https://learn.microsoft.com/en-us/azure/architecture/ (дата звернення: 10.05.2026).
17. ChurchCRM — Free Open Source Church Management System. — URL: https://churchcrm.io/ (дата звернення: 10.05.2026).
18. Rock RMS — Open Source Church Management Software. — URL: https://www.rockrms.com/ (дата звернення: 10.05.2026).
19. Sacreva - Church And Religious React JS Template. — ThemeForest. — URL: https://themeforest.net/item/sacreva-church-and-religious-react-js-template/33711719 (дата звернення: 10.05.2026).
20. Golden-Ogbeka. Church Website Template with NextJS, Tailwind and Redux Toolkit. — GitHub. — URL: https://github.com/Golden-Ogbeka/church-website (дата звернення: 10.05.2026).
21. Volik N. Digital Transformation of Ukrainian Churches: Case Studies of the Ukrainian Greek Catholic Church and the Orthodox Church of Ukraine // Religious and Digital Culture. — 2026.
22. Shevchuk D. Ukrainian Orthodoxy in the Digital Era: Building the Church Online // The New Journal of Social Research. — 2024.
23. Lomachinska I. Religious Organizations on Social Media: Opportunities, Challenges and Future Prospects for Spiritual Communities in Ukraine // SKHID. — 2025.
24. Building Scalable .NET 8 Web API: Clean Architecture + CQRS. — DEV Community. — URL: https://dev.to/iamcymentho/building-a-scalable-net-8-web-api-clean-architecture-cqrs-jwt-postgresql-redis-a-5bpj (дата звернення: 10.05.2026).

25. Code with Mukesh. CQRS with MediatR in ASP.NET Core — Complete Guide. — URL: https://codewithmukesh.com/blog/cqrs-and-mediatr-in-aspnet-core/ (дата звернення: 10.05.2026).
26. ChurchApps. CHUMS — Open Source Church Management Software (React). — GitHub. — URL: https://github.com/ChurchApps/ChumsApp (дата звернення: 10.05.2026).
27. Planning Center — Church Management Software. — URL: https://www.planning.center/ (дата звернення: 09.05.2026).
28. Tithe.ly — Church Management & Giving Platform. — URL: https://get.tithe.ly/ (дата звернення: 09.05.2026).
29. Nielsen J. Usability Engineering. — Morgan Kaufmann, 1993.
30. Krug S. Don't Make Me Think, Revisited. — New Riders, 2014.
31. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — Prentice Hall, 2017.
32. Vernon V. Implementing Domain-Driven Design. — Addison-Wesley, 2013.
33. Gamma E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. — Addison-Wesley, 1994.
34. WCAG 2.2 — Web Content Accessibility Guidelines. — URL: https://www.w3.org/TR/WCAG22/ (дата звернення: 10.05.2026).
35. Bootstrap 5 Official Documentation. — URL: https://getbootstrap.com/ (дата звернення: 10.05.2026).
36. SCSS / Sass Documentation. — URL: https://sass-lang.com/ (дата звернення: 10.05.2026).
37. GitHub Docs — Best practices. — URL: https://docs.github.com/ (дата звернення: 10.05.2026).
38. State of JS 2024 Report. — URL: https://2024.stateofjs.com/ (дата звернення: 08.05.2026).
39. REST API Design Best Practices. — URL: https://restfulapi.net/ (дата звернення: 10.05.2026).

40. Church Management System with ASP.NET. — GitHub Repository. — URL: https://github.com/AlexCode225/ASP.net-ChurchWEBAPP- (дата звернення: 10.05.2026).

ДОДАТКИ

GitHub Repo(Frontend):

https://github.com/Hvist-enota/ProjectX

GitHub Repo(Backend):

https://github.com/Hvist-enota/term-paper_church