**Звіт з розробки та затвердження архітектури проекту LambdaChess**

**Виконали**

Бенько Володимир  
Бучко Роман  
Долуда Андрій  
Зозуля Олег

**Знання та вибір підходу до побудови архітектури проекту**

При розробці проекту LambdaChess було проведено детальний аналіз наступних архітектурних підходів:

**Layered architecture**

**Layered architecture** (багатошарова архітектура) — це найбільш традиційний архітектурний патерн, де компоненти системи організовані у горизонтальні шари, кожен з яких виконує певну роль в додатку.

**Ключові особливості:**

* **Чітка ієрархія**: кожен шар має доступ лише до нижчих шарів.
* **Простота**: легка для розуміння та реалізації.
* **Структурованість**: чітка організація коду.

**Типові шари:**

1. **Presentation Layer** — відповідає за взаємодію з користувачем.
2. **Business Logic Layer** — містить бізнес-логіку додатка.
3. **Data Access Layer** — відповідає за доступ до даних.
4. **Database Layer** — сама база даних.

**Переваги:**

* Зрозуміла структура для розробників
* Простота в реалізації
* Легкість навчання нових членів команди

**Недоліки:**

* Жорстка зв'язаність між шарами
* Складність ізоляції компонентів для тестування
* Нижні шари не "знають" про верхні

**Hexagonal architecture**

**Hexagonal architecture** (портова-адаптерна архітектура) — архітектурний підхід, запропонований Алістером Кокберном, що фокусується на ізоляції ядра додатка від зовнішніх систем.

**Ключові особливості:**

* **Портова-адаптерна модель**: ядро додатка взаємодіє із зовнішнім світом через порти та адаптери.
* **Незалежність від інфраструктури**: бізнес-логіка не залежить від конкретних технологій.

**Основні компоненти:**

1. **Ядро додатка** (Домен) — містить бізнес-логіку та бізнес-правила.
2. **Порти** — визначають інтерфейси взаємодії з ядром.
3. **Адаптери** — реалізують порти для конкретних технологій.

**Переваги:**

* Висока тестованість
* Незалежність від зовнішніх систем
* Гнучкість при зміні технологій

**Недоліки:**

* Більше класів та інтерфейсів
* Складніше для розуміння початківцями
* Вимагає більше початкових зусиль при розробці

**Onion architecture**

**Onion architecture** (цибулева архітектура) — архітектурний шаблон, запропонований Джеффрі Палермо, подібний до Hexagonal, але більш деталізований.

**Ключові особливості:**

* **Принцип залежностей**: всі залежності направлені до центру.
* **Шарувата структура**: система організована у концентричні шари.

**Основні шари (від центру до периферії):**

1. **Domain Model** — ядро додатка, містить доменні об'єкти.
2. **Domain Services** — сервіси, що оперують доменними об'єктами.
3. **Application Services** — координують активності між різними сервісами.
4. **Infrastructure** — зовнішній шар для взаємодії з БД, UI, зовнішніми сервісами.

**Переваги:**

* Висока тестованість
* Зменшена зв'язаність компонентів
* Незалежність від конкретних технологій

**Недоліки:**

* Складність в розумінні та реалізації
* Більше класів та інтерфейсів
* Потребує високого рівня абстракцій

**Clean architecture**

**Clean architecture** — підхід, розроблений Робертом Мартіном, який об'єднує ідеї з Hexagonal, Onion та інших архітектур.

**Ключові особливості:**

* **Незалежність від фреймворків**: архітектура не залежить від зовнішніх бібліотек.
* **Тестованість**: бізнес-правила можуть тестуватися без UI, БД, веб-сервера.
* **Незалежність від UI**: інтерфейс можна змінювати без зміни решти системи.
* **Незалежність від БД**: бізнес-правила не прив'язані до конкретної БД.
* **Незалежність від зовнішніх агентів**: бізнес-правила не знають про зовнішній світ.

**Основні шари:**

1. **Entities** — бізнес-об'єкти додатка.
2. **Use Cases** — прикладні бізнес-правила.
3. **Interface Adapters** — конвертація даних між зовнішнім світом і use cases.
4. **Frameworks & Drivers** — інструменти та фреймворки: БД, веб-фреймворки тощо.

**Переваги:**

* Максимальна незалежність від зовнішніх систем
* Висока тестованість
* Легко заміняти компоненти системи

**Недоліки:**

* Складність в розумінні та реалізації
* Потребує більше часу на початкову розробку
* Надмірна абстракція для маленьких проектів

**Обґрунтування вибору**

Після аналізу різних архітектурних підходів, для проекту LambdaChess було обрано **Clean Architecture** з наступних причин:

1. **Масштабованість**: LambdaChess планується як довготривалий проект з потенціалом для розширення, Clean Architecture забезпечує легку масштабованість.
2. **Тестованість**: Надзвичайно важлива в проекті онлайн-гри, де логіка шахових правил повинна бути надійною та захищеною від помилок.
3. **Незалежність від технологій**: Можливість у майбутньому змінювати UI (наприклад, додати мобільні додатки) або базу даних без переписування бізнес-логіки.
4. **Модульність**: Дозволяє різним розробникам працювати над різними частинами системи незалежно.
5. **Підтримуваність**: Чітка структура полегшує підтримку та розвиток проекту в довгостроковій перспективі.

Водночас, для зменшення складності реалізації, в проекті ми дотримуємося не чистого Clean Architecture, а його адаптації під ASP.NET Core, зі збереженням ключових принципів.

**Побудова базових архітектурних структур**

Відповідно до вибраного архітектурного підходу (Clean Architecture) та проаналізувавши структуру файлів проекту, було розроблено наступну архітектурну структуру для проекту LambdaChess:

1. **Core / Domain Layer**
   * LambdaChess.DAL.Models — містить доменні моделі та абстракції
     + Abstractions/IModel.cs — базовий інтерфейс для всіх моделей
     + Enums/Winner.cs — перелік можливих результатів гри
     + GameSession.cs — модель сесії гри
     + User.cs — модель користувача
2. **Application Layer**
   * LambdaChess.BLL.Services — містить сервіси для бізнес-логіки
     + Hosting/WebApplicationBuilderExtensions.cs — розширення для конфігурації сервісів
3. **Infrastructure Layer**
   * LambdaChess.DAL.Repositories.Abstractions — абстракції для доступу до даних
     + IRepository.cs — базовий інтерфейс репозиторію
     + IUserRepository.cs — інтерфейс для роботи з користувачами
     + IGameSessionRepository.cs — інтерфейс для роботи з ігровими сесіями
   * LambdaChess.DAL.Repositories.Implementations — реалізації репозиторіїв
     + BaseRepository.cs — базова реалізація репозиторію
     + UserRepository.cs — репозиторій для роботи з користувачами
     + GameSessionRepository.cs — репозиторій для роботи з ігровими сесіями
     + Persistance/ApplicationDbContext.cs — контекст бази даних
4. **Presentation Layer**
   * LambdaChess.Web.Controllers — окремий проект для контролерів
     + BaseController.cs — базовий контролер
     + LobbyController.cs — контролер для лобі
   * LambdaChess.Web.UI — основний проект користувацького інтерфейсу
     + Controllers/GameController.cs — контролер для ігрового процесу
     + Controllers/HomeController.cs — головний контролер
     + Hubs/GameHub.cs — SignalR хаб для комунікації в реальному часі
     + Views/ — представлення MVC
     + wwwroot/ — статичні файли

Така структура забезпечує дотримання ключових принципів Clean Architecture:

* Залежності направлені до центру (доменної моделі)
* Бізнес-логіка ізольована від деталей UI та БД
* Використання інтерфейсів для абстрагування від конкретних реалізацій
* Чітке відділення презентаційної логіки від бізнес-логіки

Для забезпечення правильних залежностей між шарами було налаштовано посилання проектів:

* LambdaChess.Web.UI → LambdaChess.BLL.Services + LambdaChess.DAL.Repositories.Implementations
* LambdaChess.BLL.Services → LambdaChess.DAL.Models + LambdaChess.DAL.Repositories.Abstractions
* LambdaChess.DAL.Repositories.Implementations → LambdaChess.DAL.Repositories.Abstractions
* LambdaChess.DAL.Repositories.Abstractions → LambdaChess.DAL.Models

Це відображено в файлі рішення .sln та в файлах проектів .csproj, де вказані посилання на відповідні проекти.

**Створення бази даних**

При виборі бази даних для проекту LambdaChess було проаналізовано чотири основні варіанти:

**MS SQL Server**

**Опис**: Реляційна СУБД від Microsoft, тісно інтегрована з екосистемою .NET.

**Переваги**:

* Повна інтеграція з Entity Framework
* Висока продуктивність та надійність
* Підтримка від Microsoft
* Широкі можливості масштабування (Always On, реплікація, шардинг)
* Розширена безпека та контроль доступу

**Недоліки**:

* Високі ліцензійні витрати для комерційного використання
* Вимагає потужних серверних ресурсів
* Менш гнучкий, ніж NoSQL рішення

**MySQL**

**Опис**: Відкрита реляційна СУБД, популярна для веб-додатків.

**Переваги**:

* Безкоштовна для більшості випадків використання
* Добре інтегрується з Entity Framework Core
* Висока швидкість читання даних
* Широка спільнота та підтримка
* Легкість у встановленні та налаштуванні

**Недоліки**:

* Менш продуктивна для складних запитів, ніж MS SQL
* Менш розвинені інструменти для резервного копіювання та відновлення
* Обмежена підтримка транзакційних механізмів

**PostgreSQL**

**Опис**: Потужна відкрита об'єктно-реляційна СУБД з акцентом на розширюваність та стандарти.

**Переваги**:

* Безкоштовна та відкрита
* Потужні можливості для складних даних (JSON, масиви, гео-дані)
* Висока відповідність стандартам SQL
* Добра масштабованість
* Відмінна підтримка Entity Framework Core

**Недоліки**:

* Вища складність налаштування, ніж MySQL
* Менш широко використовується в екосистемі .NET
* Може бути повільнішою для простих операцій

**MongoDB**

**Опис**: NoSQL СУБД, що зберігає дані у форматі, подібному до JSON.

**Переваги**:

* Гнучка схема даних
* Висока продуктивність для операцій читання
* Добре масштабується горизонтально
* Зручна для зберігання складних структур даних
* Нативна підтримка JSON

**Недоліки**:

* Обмежена підтримка транзакцій
* Менша гарантія цілісності даних
* Обмежені можливості для складних запитів і з'єднання даних
* Вимагає іншого підходу до моделювання даних

**Обґрунтування вибору**

На основі аналізу структури файлів проекту LambdaChess та використаного коду, видно, що було обрано **SQLite** як базу даних для проекту. Це рішення має свої обґрунтовані переваги:

1. **Легкість розгортання**: SQLite не вимагає окремого сервера БД, що спрощує розробку та тестування.
2. **Портативність**: Файлова база даних легко переноситься між середовищами розробки.
3. **Інтеграція з Entity Framework Core**: Повна підтримка EF Core для SQLite.
4. **Достатня продуктивність**: Для проекту з помірним навантаженням (шахова гра) продуктивності SQLite цілком достатньо.
5. **Низькі системні вимоги**: Ідеально для розробки та тестування на локальних машинах.

Вибір SQLite можна розглядати як тимчасове рішення для етапу розробки, з можливістю переходу на більш потужну СУБД (наприклад, PostgreSQL) при масштабуванні проекту. Завдяки використанню Entity Framework Core та патерну Repository, зміна СУБД не вимагатиме значних змін у коді додатка.

В проекті файл бази даних SQLite знаходиться за шляхом: LambdaChess.Web.UI/app.db, а конфігурація контексту БД — у файлі LambdaChess.DAL.Repositories.Implementations/Persistance/ApplicationDbContext.cs.

**Підключення ORM**

Для проекту LambdaChess було обрано **Entity Framework Core** як ORM (Object-Relational Mapping) систему для роботи з базою даних.

**Аналіз реалізації**:

1. **Контекст бази даних**:
   * У файлі LambdaChess.DAL.Repositories.Implementations/Persistance/ApplicationDbContext.cs визначено клас, що успадковується від DbContext.
   * Контекст містить набори даних (DbSet) для основних сутностей: Users та GameSessions.
2. **Моделі**:
   * Моделі визначені в проекті LambdaChess.DAL.Models:
     + User.cs — модель користувача
     + GameSession.cs — модель ігрової сесії
   * Моделі реалізують інтерфейс IModel, що забезпечує наявність ідентифікатора.
3. **Конфігурація**:
   * Налаштування підключення до бази даних SQLite в LambdaChess.Web.UI/appsettings.json.
   * Реєстрація контексту в DI-контейнері в методі ConfigureServices класу Startup або Program.cs.
4. **Міграції**:
   * Наявні міграції в папці LambdaChess.Web.UI/Migrations:
     + 20250506073117\_Initial.cs — початкова міграція
     + ApplicationDbContextModelSnapshot.cs — актуальний знімок моделі БД
5. **Репозиторії**:
   * Патерн Repository реалізований через абстракції в LambdaChess.DAL.Repositories.Abstractions:
     + IRepository.cs — базовий інтерфейс для CRUD-операцій
     + IUserRepository.cs, IGameSessionRepository.cs — спеціалізовані інтерфейси
   * Конкретні реалізації в LambdaChess.DAL.Repositories.Implementations:
     + BaseRepository.cs — базова реалізація з використанням EF Core
     + UserRepository.cs, GameSessionRepository.cs — спеціалізовані реалізації

**Переваги використання EF Core в проекті**:

1. **Інтеграція з ASP.NET Core**: EF Core тісно інтегрований з екосистемою ASP.NET Core.
2. **LINQ-запити**: Можливість використовувати LINQ для запитів до бази даних, що підвищує продуктивність розробки.
3. **Code-First підхід**: Моделі визначаються в коді, а міграції бази даних генеруються автоматично.
4. **Підтримка різних СУБД**: Можливість легко змінити СУБД (з SQLite на PostgreSQL, MS SQL тощо) без зміни бізнес-логіки.
5. **Unit of Work**: Вбудований патерн Unit of Work через контекст бази даних.

**Особливості реалізації**:

* **Відокремлення інтерфейсів від реалізації**: Забезпечує слабку зв'язаність та легке тестування.
* **Використання асинхронних методів**: Всі операції з базою даних асинхронні (Task<T>), що покращує масштабованість.
* **Інтеграція з DI**: Залежності (репозиторії, контекст БД) впроваджуються через DI-контейнер.
* **Обробка винятків**: Використання типу Task<Result<T>> для обробки помилок у репозиторіях.

Такий підхід до реалізації ORM дозволяє досягти балансу між продуктивністю розробки, гнучкістю архітектури та продуктивністю додатка.

**Налаштування структурного логування**

У проекті LambdaChess було налаштовано структурне логування з використанням бібліотеки **Serilog** та інструменту **Seq** для зберігання та аналізу логів.

**Структурне логування** відрізняється від звичайного тим, що зберігає не лише текстові повідомлення, але й структуровані дані (об'єкти, властивості), що значно покращує можливості аналізу та фільтрації логів.

**Реалізація в проекті**:

1. **Підключення пакетів**:
   * Serilog.AspNetCore — інтеграція Serilog з ASP.NET Core
   * Serilog.Sinks.Console — вивід логів у консоль
   * Serilog.Sinks.File — запис логів у файл
   * Serilog.Sinks.Seq — відправка логів у Seq
   * Serilog.Enrichers.Environment — збагачення логів інформацією про середовище
2. **Конфігурація Serilog**:
   * Налаштування в Program.cs або через appsettings.json
   * Додавання різних "поглиначів" (sinks) для логів: консоль, файл, Seq
   * Налаштування рівнів логування для різних частин додатка
3. **Налаштування Seq**:
   * Запуск Seq як Docker-контейнера (зазначено в compose.yaml)
   * Конфігурація URL для Seq у налаштуваннях Serilog
   * Налаштування збереження та ротації логів
4. **Використання в коді**:
   * Впровадження ILogger через DI
   * Структурне логування в контролерах та сервісах
   * Логування винятків та важливих бізнес-подій

**Приклад налаштування Serilog**:

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.UseSerilog((context, services, configuration) => configuration

.ReadFrom.Configuration(context.Configuration)

.ReadFrom.Services(services)

.Enrich.FromLogContext()

.Enrich.WithMachineName()

.WriteTo.Console()

.WriteTo.File("logs/lambdachess-.log", rollingInterval: RollingInterval.Day)

.WriteTo.Seq("http://seq:5341"))

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

**Приклад конфігурації в appsettings.json**:

{

"Serilog": {

"MinimumLevel": {

"Default": "Information",

"Override": {

"Microsoft": "Warning",

"System": "Warning"

}

},

"WriteTo": [

{

"Name": "Console"

},

{

"Name": "File",

"Args": {

"path": "logs/lambdachess-.log",

"rollingInterval": "Day"

}

},

{

"Name": "Seq",

"Args": {

"serverUrl": "http://seq:5341"

}

}

],

"Enrich": [ "FromLogContext", "WithMachineName" ]

}

}

**Переваги використання Serilog і Seq**:

1. **Структуровані дані**: Можливість логувати та аналізувати не лише текст, але й структуровані об'єкти.
2. **Гнучка конфігурація**: Можливість налаштування різних рівнів логування для різних компонентів.
3. **Різноманітні "поглиначі"**: Одночасне логування в консоль, файл, Seq, базу даних тощо.
4. **Агрегація та аналіз**: Seq надає потужний інтерфейс для пошуку та аналізу логів.
5. **Моніторинг у реальному часі**: Можливість відстежувати логи в реальному часі через веб-інтерфейс Seq.

Використання структурного логування з Serilog та Seq значно полегшує відлагодження, моніторинг та аналіз поведінки додатка в різних середовищах, що особливо важливо для онлайн-гри, де критично швидко реагувати на проблеми та аналізувати поведінку користувачів.

**Висновки**

У процесі розробки архітектури проекту LambdaChess було прийнято ряд обґрунтованих рішень, які створили міцну основу для розвитку додатка:

1. **Архітектурний підхід**: Обрано Clean Architecture як найбільш відповідний для проекту онлайн-шахів. Це забезпечує тестованість, масштабованість та гнучкість у довгостроковій перспективі.
2. **Структура проекту**: Реалізовано чітке розділення на шари (Domain, Application, Infrastructure, Presentation), що відповідає принципам Clean Architecture та адаптовано під особливості ASP.NET Core.
3. **База даних**: Для розробки обрано SQLite як легку в налаштуванні та портативну СУБД, з можливістю майбутньої міграції на більш потужну базу даних.
4. **ORM**: Використано Entity Framework Core з реалізацією патерну Repository, що забезпечує абстрагування бізнес-логіки від деталей доступу до даних та спрощує тестування.
5. **Логування**: Налаштовано структурне логування з використанням Serilog та Seq, що забезпечує потужні можливості для моніторингу, відлагодження та аналізу поведінки додатка.

Обрані рішення забезпечують балансу між:

* **Якістю коду** — чиста архітектура та патерни проектування
* **Швидкістю розробки** — використання сучасних фреймворків та інструментів
* **Масштабованістю** — можливість легко розширювати функціональність
* **Підтримуваністю** — чітка структура та відокремлення відповідальностей

Ці архітектурні рішення створюють міцний фундамент для подальшого розвитку проекту LambdaChess, забезпечуючи можливість додавання нових функцій без суттєвої реструктуризації коду та ефективного масштабування системи в міру зростання кількості користувачів.