Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

УПРАВЛІННЯ ТРАНЗАКЦІЯМИ В POSTGRESQL

Текстова частина до курсової роботи

за спеціальністю «Інженерія програмного забезпечення» - 121

Керівник курсової роботи

ас. Яремко С. А.

_____

(підпис)

“___” _____

2024 року

Виконав студент

ІПЗ-3 Яценко М. О.

“___” _____ 2024 року

Київ 2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики,

проф., д.ф.-м.н.

_____М. М. Глибовець

(підпис)

„____” _____2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Яценко Марині Олексіївні факультету інформатики 3 курсу

ТЕМА: **Управління транзакціями в PostgreSQL**

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Календарний план

Вступ

1 Огляд транзакцій в PostgreSQL

2 Дизайн автоматизованої інформаційної системи

3 Розробка автоматизованої інформаційної системи

Висновки

Список використаної літератури

Дата видачі „___” _____ 2024 р. Керівник _____(підпис)

Завдання отримав_____(підпис)

Календарний план виконання роботи

№	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Визначення теми курсової роботи	08.10.2023	
2.	Ознайомлення з предметною областю	Листопад 2023	
3.	Огляд технічної літератури за темою роботи	Листопад - Грудень 2023	
4.	Початок написання теоретичної частини	Лютий 2024	
5.	Визначення основних вимог до застосунку	Березень 2024	
6.	Пошук, аналіз та вибір технологій розробки	Березень 2024	
7.	Початок створення програмної реалізації	Квітень 2024	
8.	Завершення практичної та теоретичної частин, оформлення презентації	Травень 2024	
9.	Захист курсової роботи	24.05.2024	

Студент _____

Керівник _____ “_____”_____ 2024

Зміст

Анотація

В цій курсовій роботі детально розглядаються засоби та нюанси управління транзакціями в СКБД PostgreSQL, а також процес створення автоматизованої інформаційної системи по управлінню шаховим вебзастосунком і вирішення проблем паралельного доступу до даних в цій системі. Метою роботи було здобуття навичок управління транзакціями в PostgreSQL. Результати були досягнені такими методами: опрацювання технічної літератури та формування власних висновків щодо принципів керування транзакціями; розробка власної АІС для закріплення практичних навичок.

Ключові слова: PostgreSQL, транзакції, бази даних, ACID, MVCC, багатоверсійність, рівні ізольованості, Java, JSP, servlets.

Вступ

У 21 столітті інформаційні системи відіграють важливу роль у повсякденному житті людей та бізнесу. Вони забезпечують безпечне зберігання, обробку та вилучення величезних обсягів даних, які є ключовими для працездатності цих систем. Але кількість та складність даних, які необхідно обробляти, постійно зростає, через що виникає необхідність в ефективних механізмах управління ними та забезпеченні їхньої стабільної роботи.

У цій роботі буде окреслено суть поняття транзакції та зазначено, чому для досягнення вищезгаданої мети важливе використання транзакцій; буде розглянуто принципи управління транзакціями в контексті PostgreSQL, однієї з провідних відкритих СКБД. Завдяки її надійності, продуктивності та гнучкості PostgreSQL широко використовують в різних галузях, включаючи веб-розробку. Вона надає потужні можливості для керування транзакціями, відповідає принципам ACID, а також вирішує одну з найскладніших проблем будь-якої СКБД — керування паралельним доступом до даних — за допомогою впровадження багатоверсійності. У роботі буде розглянуто особливості механізму багатоверсійності в PostgreSQL та його переваги над традиційним механізмом блокування.

Окрім огляду теоретичних основ, в цій роботі буде продемонстровано практичну реалізацію автоматизованої інформаційної системи у вигляді шахового вебзастосунку з використанням паралельних транзакцій для наочного вирішення конфліктів паралельного доступу.

Розділ 1. Огляд транзакцій в PostgreSQL

1.1. Поняття транзакції

Сучасні СКБД надають користувачам безліч можливостей для управління даними. Одна з них — використання транзакцій, що є невід'ємним компонентом для успішного керування будь-якою базою даних, якщо адміністратор БД прагне забезпечити консистентність та узгодженість збереженої інформації.

Класичним прикладом недотримання цілісності даних є переказ коштів з одного банківського рахунку на інший. Що буде, якщо кошти успішно спишуться з *рахунку 1,* але під час нарахування їх на *рахунок 2* виникне помилка? Зміни в базі даних будуть непослідовні і не відображатимуть реальний стан речей — частина грошей "зникне" в нікуди. Порушується узгодженість бази даних.

Запобігти таким ситуаціям можна, використовуючи **транзакції**. Транзакція являє собою певну групу логічно пов'язаних операцій над базою даних, які мають бути успішно *виконані всі разом*, або *жодна з них не має бути виконана*. Транзакцію можна розглядати як обгортку над деякою послідовністю звернень до БД, що об'єднує їх в єдину одиницю роботи і забезпечує передбачувану поведінку як у разі успіху, так і в разі невдачі. Транзакції є своєрідними будівельними блоками для безпечної та послідовної модифікації даних засобами СКБД.

Якби SQL-запити з попереднього прикладу виконувались, як транзакція, помилка під час виконання другого запиту призвела би до того, що перший запит також буде скасовано, і узгодженість даних не було би порушено.

Виділяють **явні** та **неявні** транзакції. У PostgreSQL будь-які звернення до бази даних (окрім BEGIN та COMMIT) є неявними транзакціями [1]. Явні транзакції можна ініціювати за допомогою ключового слова BEGIN (або START TRANSACTION — в PostgreSQL ці операції є синонімічними, однак деякі інші СКБД не підтримують операцію BEGIN).

1.2. Фази транзакції

Розпочати транзакцію в PostgreSQL можна за допомогою методу BEGIN. У разі успішного виконання всіх операцій всередині транзакції вона фіксується командою COMMIT; у випадку виникнення помилки всі зміни, внесені транзакцією, скасовуються командою ROLLBACK.

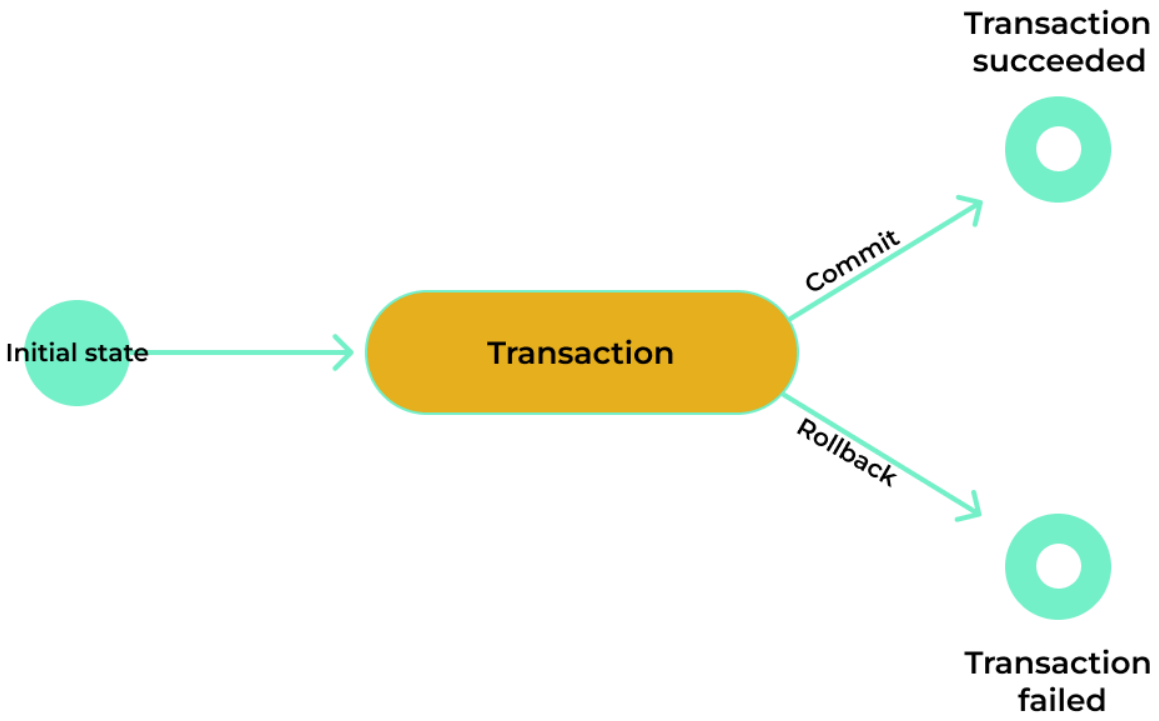

Рисунок 1.1 — Фази транзакції

Після команди BEGIN у будь-який момент часу транзакція може перебувати в одному з можливих станів [7]:

1) **Активна** — транзакція перебуває в цій фазі під час виконання запитів до БД або модифікації даних, коли ще не всі її операції було завершено;
2) **Частково зафіксована** — після завершення всіх звернень до БД в межах транзакції, внесені нею зміни фіксуються в локальному буфері або пам’яті програми. Але на цьому етапі їх ще не було внесено до бази даних командою COMMIT;
3) **Невдала** — транзакція переходить в цю фазу, якщо під час виконання її операцій або під час внесення змін, зроблених транзакцією, до бази даних було згенероване виключення;
4) **Перервана** — фаза настає відразу після попередньої — зміни скасовуються операцією ROLLBACK;
5) **Зафіксована** — у разі успішного виконання транзакції внесені зміни фіксуються в базі даних операцією COMMIT;
6) **Завершена** — ця фаза є фінальною для транзакції і настає після того, як транзакція була зафіксована командою COMMIT або скасована командою ROLLBACK.

Окрім стандартних команд керування транзакціями (BEGIN, COMMIT та ROLLBACK) PostgreSQL також підтримує **точки збереження**. За допомогою команди SAVEPOINT можна розпочати підтранзакцію в межах основної транзакції, а команда ROLLBACK TO SAVEPOINT скасує всі зміни в межах підтранзакції, не впливаючи на інші операції основної транзакції. Команда RELEASE SAVEPOINT скасує точку збереження, не відміняючи зміни в межах підтранзакції [1].

1.3. Властивості ACID

ACID — абревіатура, що позначає набір властивостей, які має мати транзакція для забезпечення надійності та коректності стану бази даних навіть у випадку виникнення помилок [6]:

1) **Атомарність** (Atomicity) — цей принцип наголошує, що транзакція має буде оброблена як єдина логічна операція. Або всі звернення до БД в межах транзакції мають бути успішно виконані, або жодне;
2) **Узгодженість** (Consistency) — характеризується тим, що як до, так і після виконання транзакції база даних має перебувати у коректному стані: наприклад, зовнішні ключі не можуть посилатись на неіснуючі записи, вік користувачів не може приймати від'ємне значення, тощо;
3) **Ізольованість** (Isolation) — згідно з цим принципом, паралельне виконання транзакцій має привести до стану даних, який можливо було б досягти за їх послідовного виконання. Дотримання цього принципу безпосередньо пов'язане з вирішенням конфліктів паралельного доступу;
4) **Найдініть** (Durability) — цей принцип гарантує, що після фіксації результатів транзакції (після операції COMMIT) зміни будуть збережені навіть у разі збою системи. Зазвичай це досягається фіксуванням наслідків транзакції в енергонезалежну пам'ять.

1.4. Проблеми (аномалії) паралельного доступу до даних

Однією з найскладніших задач для будь-якої СКБД є керування паралельним доступом до даних. Особливо актуально це для веб-додатків, адже вони можуть щосекунди приймати багато запитів на підключення і мають певним чином вирішувати проблему одночасного зчитування та модифікації даних.

Операції читання (SELECT) в основному не спричиняють аномалії при паралельній обробці транзакцій, адже є “безпечними” [ДЖЕРЕЛО]. Проблеми виникають при створенні, оновленні та видаленні даних.

У 1992 р., коли Американський інститут національних стандартів (ANSI) розробив стандарт SQL-92, було виділено три основні аномалії, що можуть виникати під час виконання паралельних транзакцій [3]:

1) **Брудне читання** (dirty reads) — виникає, коли транзакція зчитує дані, що були змінені іншою паралельною транзакцією, але ще не були зафіксовані (UNCOMMITTED). Якщо транзакцію не було підтверджено, нема гарантії, що її буде підтверджено взагалі — можливо, її буде скасовано. В такому разі першою транзакцією було б зчитано дані, які в БД ніколи насправді не містились.

Транзакція А	Транзакція Б
BEGIN;	BEGIN;
INSERT INTO user_credentials (nickname, email, password) VALUES ('test','user@ukma.edu.ua' , 'some_pass');	
	SELECT * FROM user_credentials WHERE nickname='test';
ROLLBACK;	

Таблиця 1.1 — Приклад брудного читання

У прикладі, що наведено в таблиці 1.1, Транзакція А реєструє в застосунку нового користувача з нікнеймом “test”. В цей час Транзакція Б хоче отримати дані про користувача “test”, якщо такий існує. Через те, що їй доступні зміни, внесені Транзакцією А, Транзакція Б отримає ці дані. Потім в Транзакції А виникає

помилка і вона скасовується командою ROLLBACK. Результат Транзакції Б буде містити некоректні дані, які насправді ніколи не були зафіксовані в БД.

2) **Неповторюване читання** (non-repeatable reads) — виникає, коли ідентичні запити в межах однієї транзакції повертають неоднакові результати. Це відбувається, якщо з моменту виконання першого запиту дані було оновлено деякою зафіксованою (COMMITTED) паралельною транзакцією, а далі зчитано повторним запитом.

Транзакція А	Транзакція Б
BEGIN;	BEGIN;
	SELECT country INTO user_country FROM user_details WHERE user_id=1;
	IF (user_country='Ukraine') THEN
UPDATE user_details SET country='Poland' WHERE user_id=1;	
COMMIT;	
	UPDATE user_details SET discount=10 WHERE user_id=1;
	END IF; COMMIT;

Таблиця 1.2 — Приклад неповторюваного читання

У прикладі з таблиці 1.2 Транзакція Б хоче встановити користувачу з id=1 знижку 10%, якщо він перебуває в Україні. Перший запит SELECT в Транзакції Б отримує дані про користувача, а далі оновлює йому знижку, якщо дана умова виконується. Але між першим та другим запитом SELECT користувач змінив свою країну на Польщу. В результаті він все одно отримає знижку, хоча якби Транзакція Б перевірила його країну повторно, умова для знижки вже не виконувалась би.

3) **Фантомне читання** (phantom reads) — виникає, коли транзакція виконує повторне зчитування одних і тих же даних, але в результаті отримує нові записи, яких при першому читанні не існувало. Це відбувається, якщо з моменту виконання першого запиту до БД було занесено нові дані деякою зафіксованою (COMMITTED) паралельною транзакцією.

Транзакція А	Транзакція Б
BEGIN;	BEGIN;
	SELECT * FROM user_details WHERE city='Kharkiv';
INSERT INTO user_details (user_id, city) VALUES (222, 'Kharkiv')	
COMMIT;	
	SELECT * FROM user_details WHERE city='Kharkiv';

Таблиця 1.3 — Приклад фантомного читання

У прикладі, що наведено в таблиці 1.3, Транзакція Б двічі робить запит на отримання всіх користувачів з міста Харків, але між цими запитами Транзакція А реєструє в системі нового харків’янина. Це може призвести до помилок узгодженості даних: наприклад, якщо перший запит було зроблено, щоб вивести кількість користувачів з Харкова на сторінку, а другий — щоб відобразити самих користувачів, відображена кількість не буде відповідати дійсності.

З огляду на ці аномалії комітетом ANSI було визначено “ідеал”, до якого мають прагнути СКБД: паралельне виконання транзакцій має призводити до результату, ідентичного з послідовним їх виконанням, а отже не має призвести до жодної з трьох перелічених аномалій [3]. Рівень ізольованості транзакцій, що відповідають

цим критеріям, було названо Serializable. Детальніше про рівні ізольованості в наступному підрозділі.

У 1995 р. компанією Microsoft було випущено статтю з критикою рівнів ізольованості стандарту SQL-92, що включав перелік різноманітних аномалій, не передбачуваних стандартом, а отже формально дозволених на рівні Serializable [4]:

Table 4. Isolation Types Characterized by Possible Anomalies Allowed.

Isolation level	**P0** Dirty Write	**P1** Dirty Read	**P4C** Cursor Lost Update	**P4** Lost Update	**P2** Fuzzy Read	**P3** Phantom	**A5A** Read Skew	**A5B** Write Skew
READ UNCOMMITTED == Degree 1	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
READ COMMITTED == Degree 2	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL SERIALIZABLE == Degree 3 == Repeatable Read Date, IBM, Tandem, ...	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

Рисунок 1.2 — Повний перелік аномалій паралельного доступу та рівні ізольованості, на яких вони виникають

1.5. Рівні ізольованості транзакцій

1.5.1 Чотири рівні ізольованості

Рівень ізольованості транзакції визначає, якою мірою транзакція впливає на транзакції, що виконуються паралельно, а вони — на неї [3]. Найвищим рівнем є Serializable — за визначенням, паралельне виконання транзакцій на цьому рівні

призведе до такого ж результату, як і якби ці транзакції виконувалися послідовно в деякому порядку.

У стандарті SQL-92 було затверджено **чотири** рівні ізольованості транзакцій з огляду на те, які з аномалій паралельного доступу є прийнятними на кожному рівні:

Рівні ізольованості	**Dirty reads**	**Non-repeatable reads**	**Phantom reads**
Read Uncommitted	Можливі	Можливі	Можливі
Read Committed	Неможливі	Можливі	Можливі
Repeatable Reads	Неможливі	Неможливі	Дозволено
Serializable	Неможливі	Неможливі	Неможливі

Таблиця 1.4 — Рівні ізольованості та можливі аномалії (SQL-92)

Така стандартизація рівнів ізольованості дозволяє розробникам програмного забезпечення робити зважені та обмірковані рішення про те, який саме рівень ізольованості буде оптимальним для тих чи інших транзакцій: наприклад, робити вибір на користь більшої продуктивності і обирати рівень нижче, ніж Serializable, якщо певні аномалії паралельного доступу не є критичними в контексті застосунку.

1.5.2 Імплементація рівнів ізольованості в PostgreSQL

У PostgreSQL імплементація рівнів ізольованості має деякі особливості. Аномалія брудного читання неможлива на будь-якому з рівнів ізольованості, тому рівень Read Uncommitted як такий відсутній: його можна вказати, але насправді буде застосовано рівень Read Committed. Таке рішення пояснюється тим, що це єдиний спосіб послідовно відобразити стандартизовані рівні ізольованості в архітектуру

MVCC (керування паралельним доступом за допомогою багатоверсійності), що використовується в PostgreSQL [1, розділ 13.2].

Рівні ізольованості	**Dirty reads**	**Non-repeatable reads**	**Phantom reads**	**Serialization anomalies**
Read Uncommitted	Дозволені, але не в PG	Можливі	Можливі	Можливі
Read Committed	Неможливі	Можливі	Можливі	Можливі
Repeatable Reads	Неможливі	Неможливі	Неможливі	Можливі
Serializable	Неможливі	Неможливі	Неможливі	Неможливі

Таблиця 1.5 — Рівні ізольованості та можливі аномалії (PostgreSQL)

За замовчуванням PostgreSQL використовує рівень ізольованості Read Committed. Щоб встановити інший рівень ізольованості, слід скористатись командою SET TRANSACTION.

1.5.3 Порівняння рівнів ізольованості в PostgreSQL

Часткова ізольованість транзакцій, яку забезпечує режим Read Committed, насправді є достатньою для багатьох програм.Перевагами цього режиму є його швидкість і простота в використанні. Справа в тому, що чим суворіше рівень ізольованості, тим вище вірогідність виникнення помилок під час спроби зафіксувати (COMMIT) паралельні транзакції [1].

Наприклад, якщо паралельні транзакції виконуються з рівнем ізольованості Serializable, перед її фіксуванням PostgreSQL має перевірити, чи внесені у БД

зміни можна було досягти послідовним використанням цих транзакцій. Інакше він скасує одну з них та повідомить про помилку серіалізації, а далі спробує виконати скасовану транзакцію знову. Тому використовуючи рівні ізольованості Serializable та Repeatable Read треба бути готовим до збільшеної кількості скасувань та повторів транзакцій у системі, що може негативно вплинути на швидкодію.

З огляду на це розробник має робити зважене рішення щодо того, який саме рівень ізольованості транзакцій обрати для певних операції у своєму застосунку, та робити вибір на користь вищої продуктивності або суворішої узгодженості даних.

1.6. Багатоверсійність (MVCC) в PostgreSQL

PostgreSQL керує паралельним доступом до даних за допомогою моделі **MVCC (Multiversion Concurrency Control**). На відміну від забезпечення ізоляції за допомогою механізму блокування у традиційних базах даних (коли транзакція тимчасово блокує рядок чи таблицю, з якою проводить операції, для читання чи модифікації паралельними транзакціями), PostgreSQL намагається мінімізувати блокування і натомість використовує **Snapshot Isolation** (ізоляція за допомогою знімків) [1].

Механізм полягає у зберіганні в базі даних різних версій одного і того ж запису. Коли транзакція оновлює дані, вона не змінює оригінальний запис одразу, а створює в таблиці його нову версію і працює з нею до моменту її зафіксування. Натомість, запити SELECT бачать “знімок” таблиці в момент, коли запит почав свою роботу, та працюють з ним. Кожній новій транзакція після її першої операції запису призначається унікальний **XID**, і далі вона звертається лише до версій рядків, що мають найновішу версію серед нижчих за її XID. Завдяки MVCC паралельні транзакції можуть працювати з різними версіями одного й того ж рядка та уникати конфліктів між собою.

1.7. Механізми блокування

Не зважаючи на існування MVCC, механізм блокування рядків та таблиць також реалізовано в PostgreSQL — їх можна використовувати в застосунках, які не потребують повної ізольованості транзакцій, або якщо розробник віддає перевагу явному керуванню транзакціями в точках конфліктів [1]. Явно заблокувати рядок або таблицю в PostgreSQL можна за допомогою команди LOCK.

Однак до явного блокування рекомендується звертатись лише за необхідності, тобто тоді, коли MVCC не дає бажаної поведінки. У більшості випадків MVCC надає вищу безпеку та швидкодію. Зокрема, явне блокування може призвести до виникнення **deadlocks**, або взаємного блокування — ситуації, коли декілька транзакцій чекають на ресурси, заблоковані іншими, і жодна з них не може продовжити свою роботу. PostgreSQL вирішує взаємні блокування автоматично, але не одразу — час до запуску механізму вирішення взаємних блокувань визначається параметром deadlock_timeout. Можна зробити висновок, що явне використання блокувань в PostgreSQL потребує великої обережності з боку розробника.

Розділ 2. Дизайн автоматизованої інформаційної системи

2.1. Загальна характеристика застосунку

Для здобуття практичного досвіду з управління транзакціями в PostgreSQL автором було розроблено автоматизовану інформаційну систему з підтримкою паралельних транзакцій.

Застосунок являє собою шаховий веб-сервіс, який дозволяє користувачам реєструвати облікові записи, після чого вони можуть створювати та налаштовувати запити на шахові партії, приймати виклики інших гравців. Партії зберігаються в базі даних, тому після їх завершення гравець може переглядати та аналізувати свої помилки.

Під час партії гравці можуть обмінюватись повідомленнями в чаті, а в разі порушення правил сайту гравець має можливість поскаржитись на свого опонента.

2.2. Аналоги застосунку на ринку

Для прийняття остаточних рішень щодо архітектури та функціональних можливостей, яких потребує застосунок, автором було досліджено два аналогічних сервіси, що на сьогодні є одними з найбільш популярних на ринку:

1) **Chess.com**

   Chess.com — шахова інтернет-спільнота, що поєднує в собі шаховий сервер, форум та соціальну мережу. Вебсайт дозволяє грати в шахи з іншими користувачами або ШІ, розв’язувати шахові головоломки, аналізувати зіграні партії за допомогою шахових рушіїв та виправляти свої помилки, а

також спілкуватись з гравцями і залишати коментарі на форумі. На сьогодні chess.com є лідером по кількості відвідувань серед усіх шахових вебсайтів в мережі Інтернет [8].

Рисунок 2.1 — головна сторінка вебсайту “Chess.com”

2) Lichess.org

Lichess — шаховий інтернет-сервер з багатим функціоналом. У ньому також доступні різні режими гри: класична або гра по листуванню (без обмеження в часі). Гравці мають можливість налаштовувати партію як завгодно, обираючи необхідні параметри, і запрошувати інших гравців зіграти з ними. Сайт є дуже популярним, хоча і поступається Chess.com за цим показником.

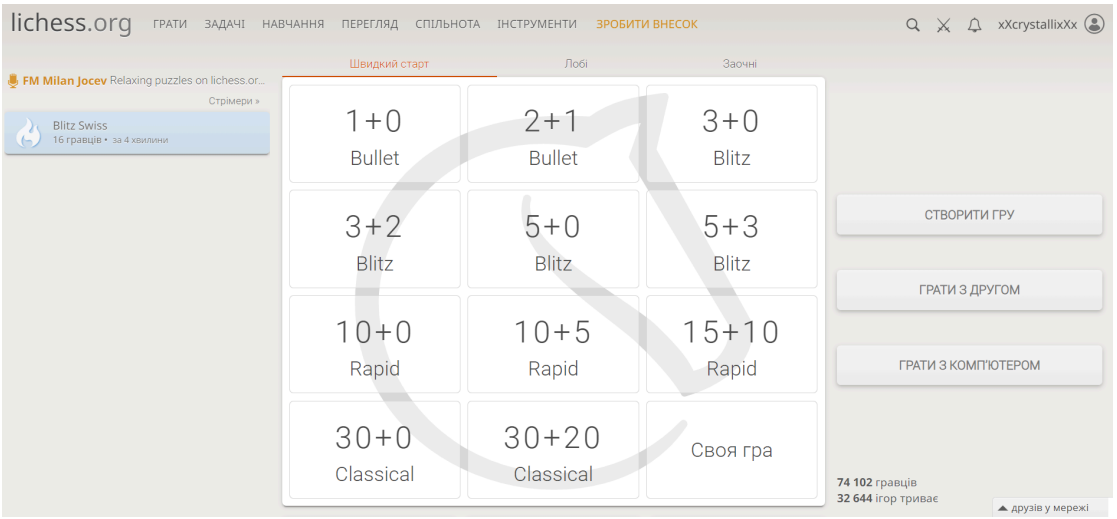

Рисунок 2.2 — головна сторінка вебсайту “Lichess”

Загалом застосунки є дуже подібними. В ході дослідження та на підґрунті власного досвіду (обидва ресурси автор використовує регулярно) було виділено такі ключові відмінності між двома застосунками:

- Lichess є повністю безкоштовним сервісом, в той час як Chess.com пропонує багато додаткових функцій для преміум-користувачів (відеоуроки, шахові пазли, інструменти для аналізу партій);
- Lichess відзначається простим користувацьким інтерфейсом з мінімальною кількістю відволікаючих елементів, віддаючи перевагу функціональності, а не естетиці. Водночас Chess.com має більш вдосконалений комерційний дизайн, роблячи акцент на залучення нових користувачів за допомогою візуалу;
- Chess.com є не лише шаховим інтернет-сервісом, а й соціальною мережею: користувачі можуть спілкуватися на форумах та в особистих повідомленнях, об’єднуватись в клуби за інтересами. Lichess також

підтримує листування, але в спрощеному варіанті, і більше зосереджений саме на шахових партіях між користувачами;
- Lichess має відкритий програмний код.

Основною ж функціональною відмінністю між **Lichess** та **Chess.com** є те, як застосунки обробляють запити користувачів на гру: на **Lichess** користувачі можуть бачити запити інших гравців в ігровому лобі та самостійно вирішувати, до якої гри приєднатися. Їм доступні всі деталі гри перед її початком, в тому числі нікнейм опонента, його рейтинг та обраний колір фігур. Натомість **Chess.com** обробляє запити на гру самостійно, призначаючи близьких за рейтингом опонентів одне одному.

У своїй системі автором було вирішено імплементувати саме перший варіант, адже ігрове лобі, до перегляду та модифікації якого багато користувачів має одночасний доступ, є потенційним джерелом конфліктів між паралельними транзакціями. Реалізація такої системи зможе дати практичний досвід у вирішенні конфліктів паралельного доступу до бази даних.

2.3. Проектування бази даних

2.3.1 Визначення основних цілей розробки бази даних

Перед створенням моделі БД було визначено автором наступні вимоги до даних, що зберігаються

- Шахова партія: мають зберігатись посилання на чорного та білого гравців, переможця (якщо гра відбулася), тип гри та часове обмеження; має бути зафіксовано рейтинг кожного гравця на початку гри, дату початку і дату закінчення партії. Кроки партії мають зберігатись в шаховій алгебраїчній нотації;

- Запит на гру: в БД мають зберігатись лише актуальні запити на гру. Після скасування запиту його має бути видалено з БД. Зберігатися має така інформація: ініціатор запиту та всі його бажані налаштування щодо партії (колір фігур, рейтинг суперника, часове обмеження, тип гри);
- Користувач: в БД мають бути збережно облікові дані користувача, а також його рейтинг по кожному типу партій (стандартне значення 1500), країна, стать, опис профілю та посилання на фотографію профілю;
- Повідомлення чату: партії передбачають можливість спілкування опонентів в чаті, тому база даних має зберігати відправника та текст кожного повідомлення, посилання на партію, в якій його було відправлено, та час відправки;
- Скарга: у разі порушення іншим гравцем правил сайту користувач має можливість поскаржитись на нього. В БД має зберігатись інформація про скаржника та оскарженого, час створення скарги, правило, що було порушене, деталі (причину) скарги а також інформацію про модератора або адміністратора, що забронював скаргу, якщо такий є.

2.3.2 Концептуальна (інфологічна) модель

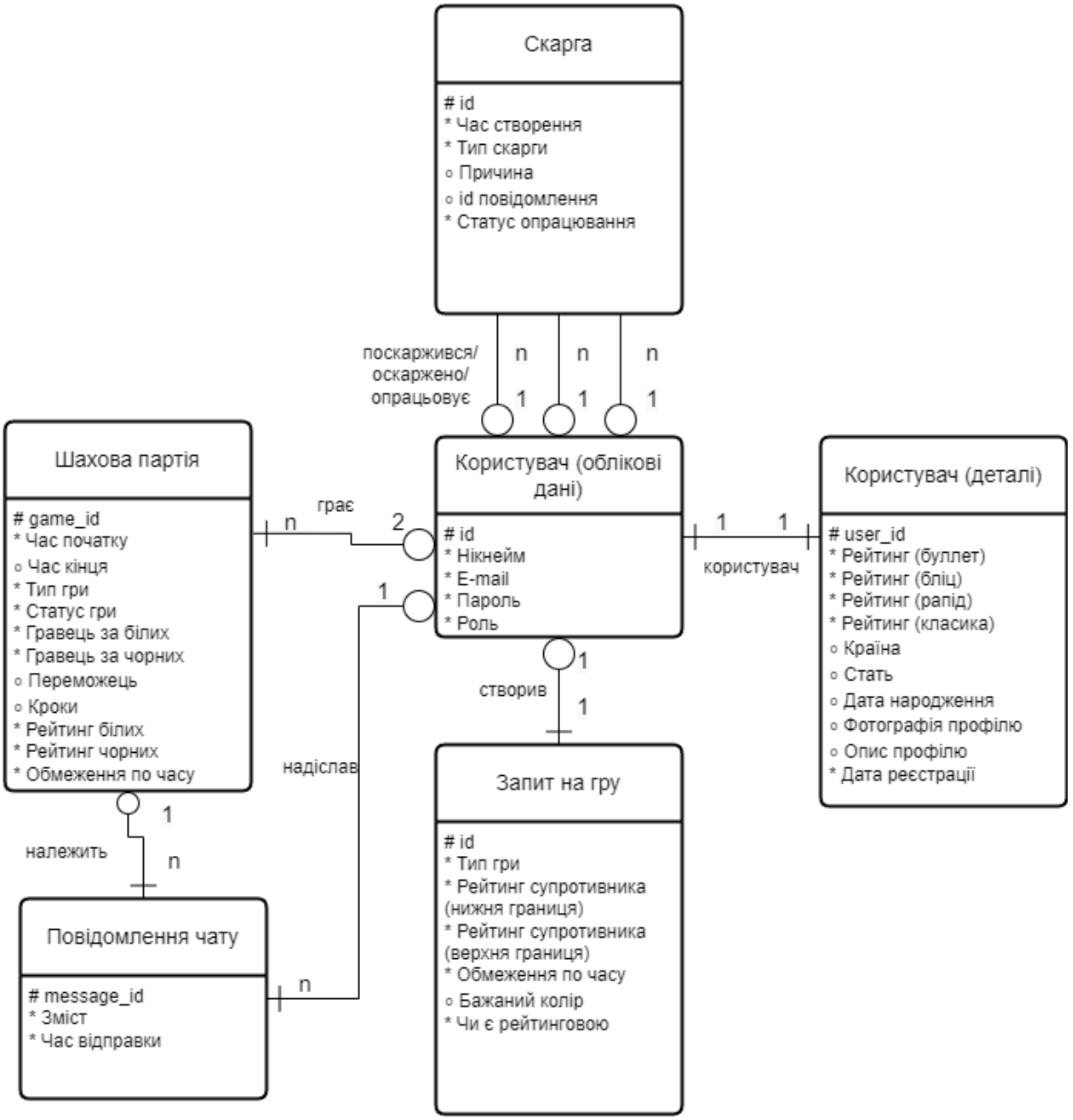

Рисунок 2.3 — ER-модель

2.3.3 Реляційна (даталогічна) модель

Див. додаток Б.

2.4. Архітектура системи

Додаток було вирішено писати на мові Java, застосувавши якості архітектурного шаблону було обрано модель MVC (Model–view–controller). При такому підході програмна логіка складається з трьох взаємопов’язаних компонентів:

- **Модель** — частина застосунку, яка повністю реалізує бізнес-логіку та звернення до бази даних. Цю частину можна також розділити на окремі рівні: на рівні **сутностей** кожній таблиці в базі даних відповідає певний класс Java; рівень **репозиторію** здійснює комунікацію з базою даних за допомогою **DAO (Data Access Object)**, а рівень **сервісів** є посередником між контролером та моделлю;
- **Представлення** — рівень, що відповідає за відображення відповіді користувачеві у формі графічного інтерфейсу (у випадку вебзастосунку це HTML-сторінка в браузері, а в даній АІС рівень реалізовано за допомогою JSP-сторінок);
- **Контролер** — модуль керування, що приймає запити від користувачів та звертається до ресурсів рівню моделі, щоб обробити їх, а також до рівню представлення, щоб коректно відобразити відповідь в браузері.

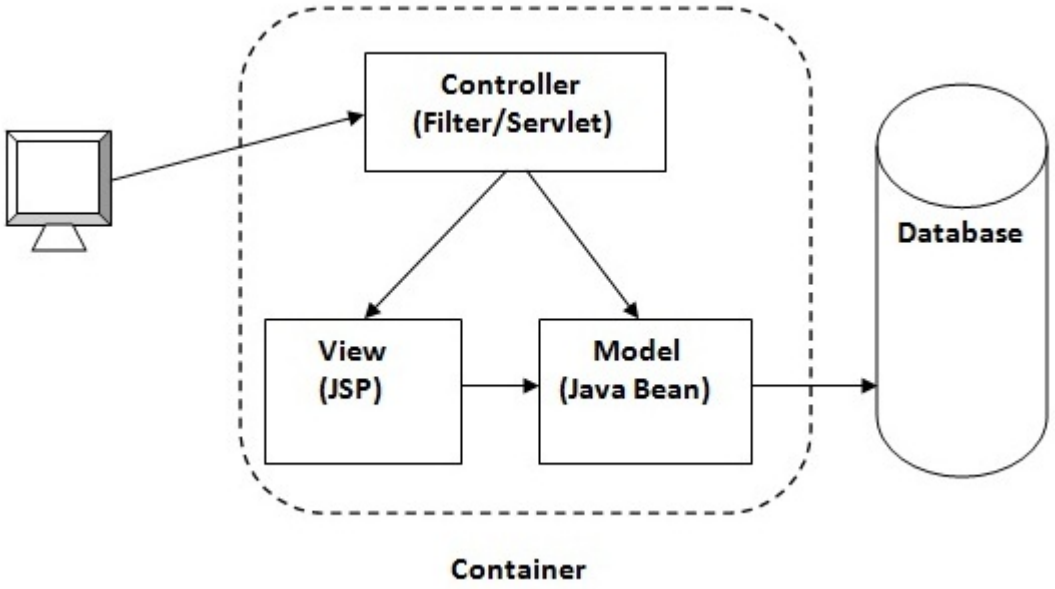

Рисунок 2.4 — схема архітектури MVC з використанням технологій JSP та сервлетів

Сьогодні шаблон MVC широко використовується в розробці програмних застосунків, адже робить структуру програмного забезпечення гнучкою, масштабованою та сприяє перевикористанню програмних компонентів [9].

2.5. Інтерфейс користувача

Більшість елементів графічного інтерфейсу розміщено на навігаційній панелі. Перш ніж користуватися можливостями застосунку, гравець має зареєструвати обліковий запис:

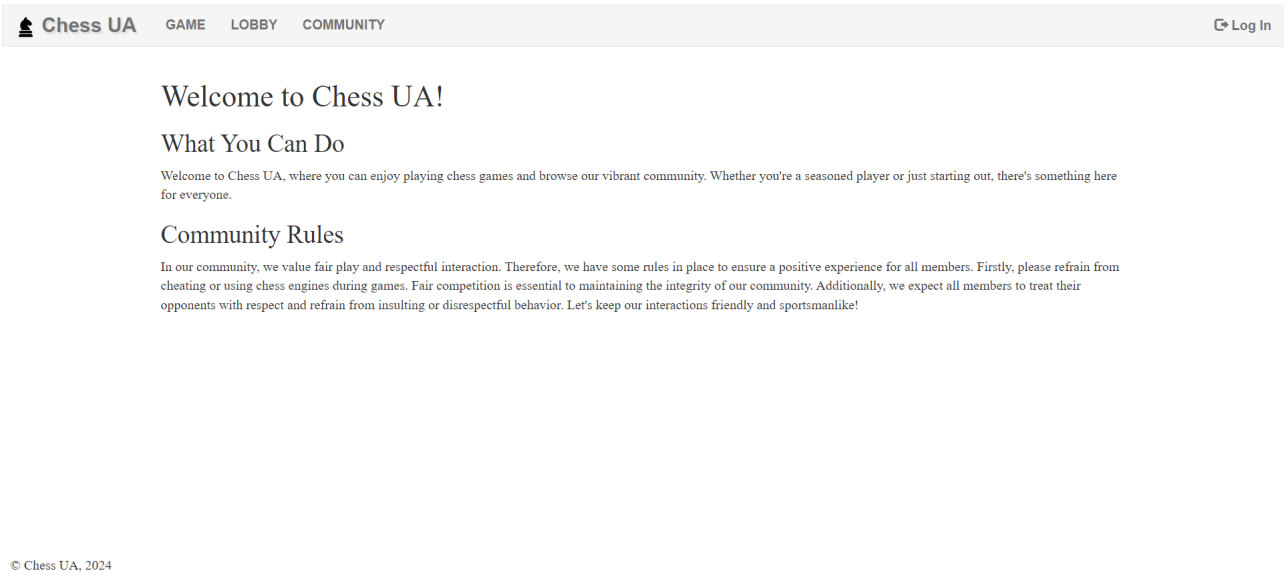

Рисунок 2.5 — Головна сторінка застосунку

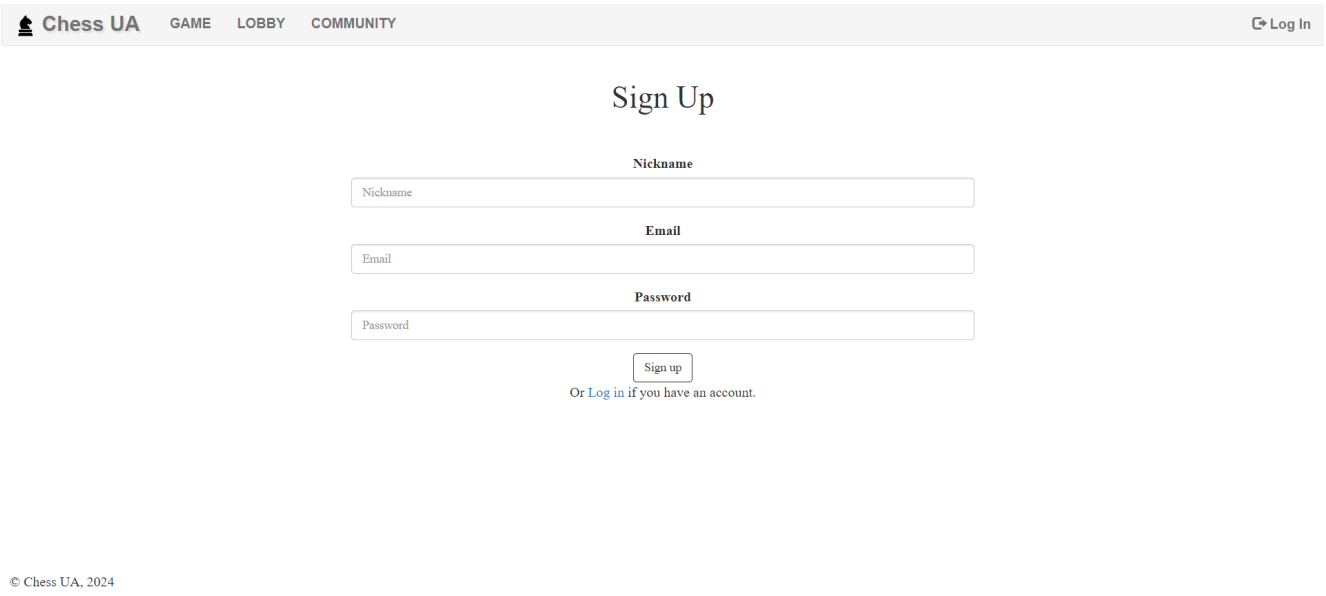

Рисунок 2.6 — Сторінка реєстрації

Авторизовані користувачі можуть створювати запити на шахові партії та обирати налаштування до них, або проглядати ігрове лобі в пошуках викликів, що їх зацікавлять:

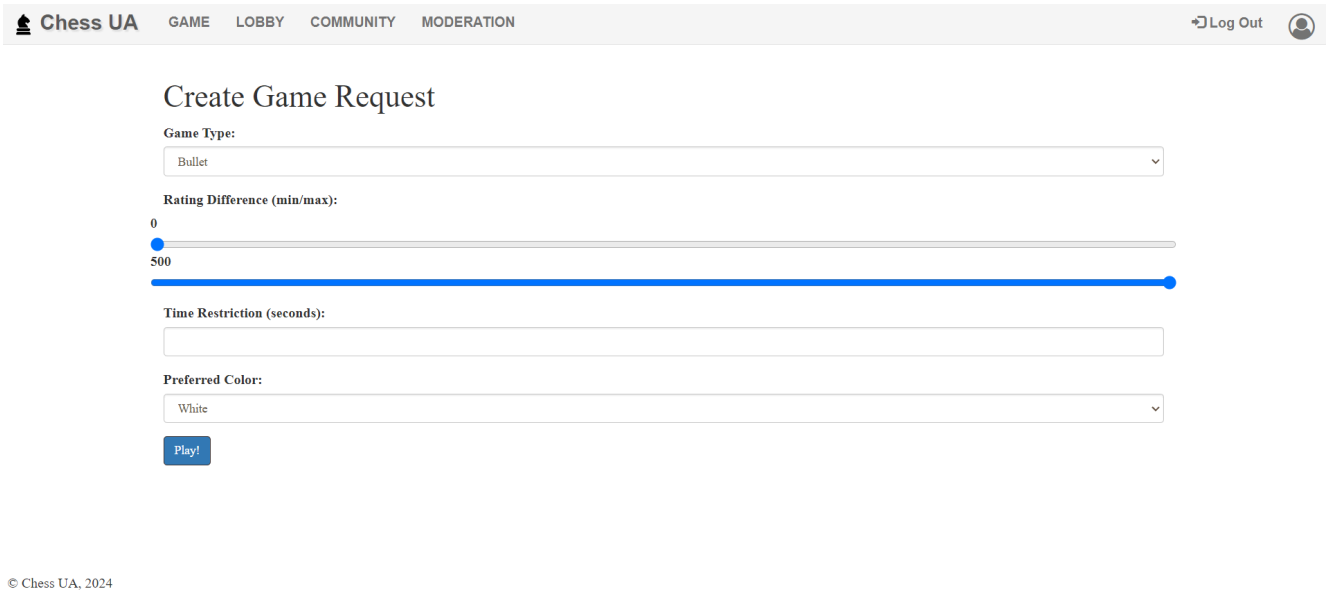

Рисунок 2.7 — Сторінка створення гри

Користувачам також доступний перегляд та редагування деталей власного профілю і перегляд профілів інших гравців. На сторінці кожного профілю користувачеві доступна кнопка “поскаржитись”, щоб повідомити модераторам про порушення іншим користувачем правил сайту:

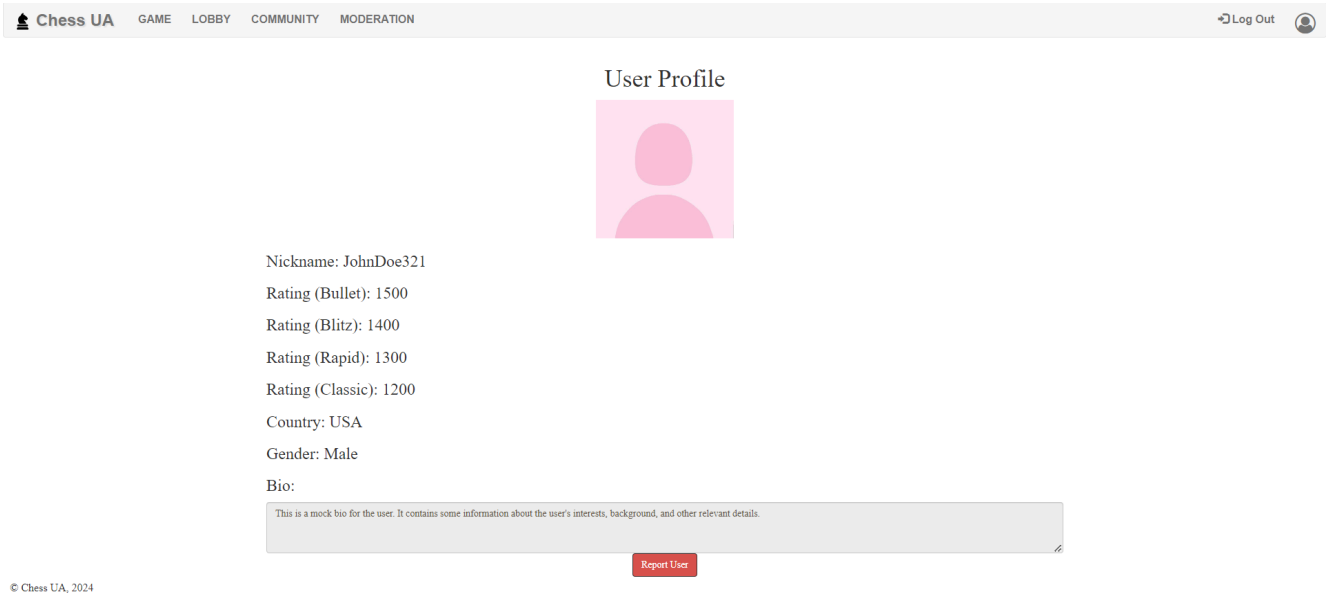

Рисунок 2.8 — Перегляд профілю

Користувачі з ролями “модератор” або “адміністратор” мають можливість займатись модерацією сайту: вручну заносити шахові партії (наприклад, які відбулися в режимі офлайн), коригувати рейтинг користувачів, видаляти їх, а також опрацьовувати скарги від гравців.

Розділ 3. Розробка автоматизованої інформаційної системи

3.1. Обрані технології

Як інструмент для збирання проекту та керування залежностями в розробці було використано **Apache Maven.** Автор зумовлює вибір популярністю та потужністю цієї платформи.

Архітектурний шаблон MVC було реалізовано з використанням **JSP (Jakarta Server Pages)** для побудови рівню представлення, класів Java для рівню моделі та **Java Servlets** для рівню контролера. Разом з JSP також було використано фреймворк **Twitter Bootstrap** для гнучкого та чутливого до пристрою користувача відображення сторінок.

У якості контейнера сервлетів було обрано **Apache Tomcat.** Вибір пояснюється тим, що даний вебсервер є нативним для мови Java. Apache Tomcat виконує роль обробки URL-запиту від клієнта та звертається до відповідного сервлету. Сервлет обробляє запит і віддає відповідь у вигляді Java-об’єкту, а контейнер серверів перетворює його на HTTP відповідь та передає користувачеві. [10]

Для роботою з базою даних PostgreSQL (створення та редагування схем, таблиць, користувацьких типів) автором було використано безкоштовний графічний клієнт

pgAdmin 4. Це рішення зумовлено його доступністю, популярністю і зручним (порівняно з аналогічним консольним застосунком psql Shell) користувацьким інтерфейсом.

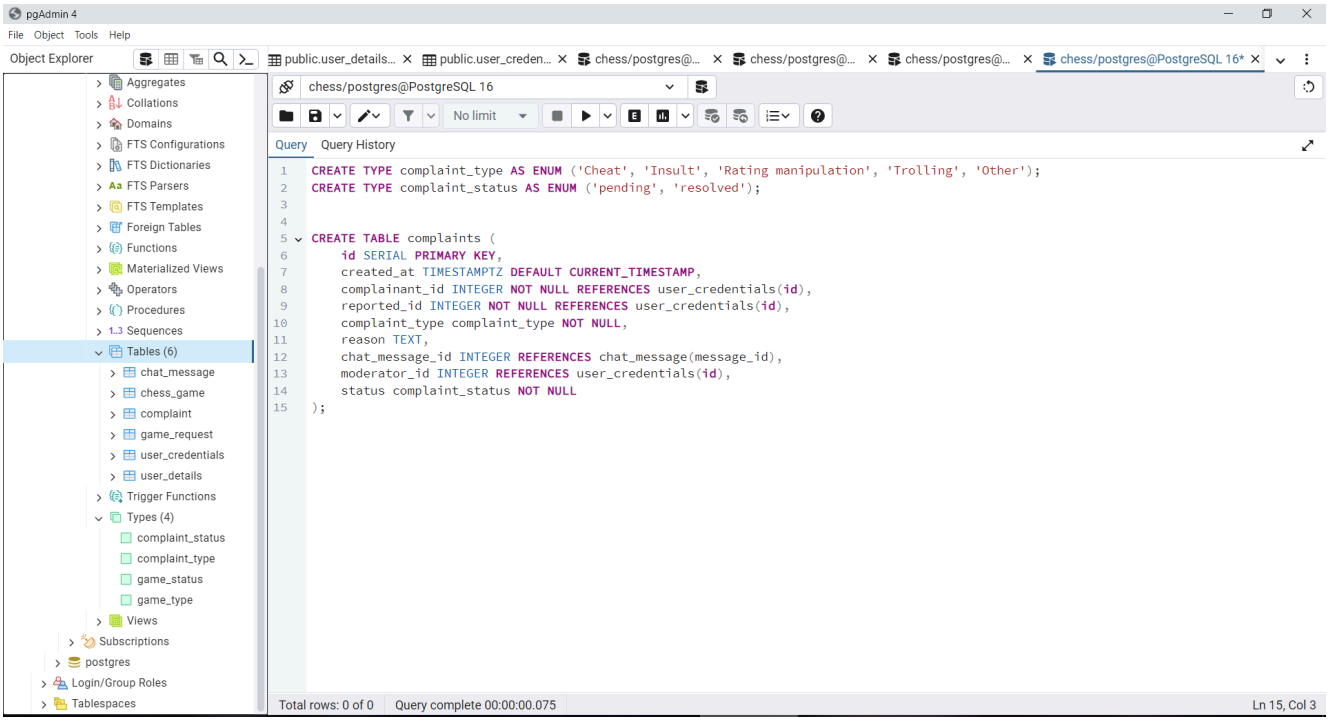

Рисунок 3.1 — створення таблиці “Скарги” засобами pgAdmin 4

Зв’язок бізнес-логіки з базою даних в застосунку здійснюється за допомогою **JDBC (Java Database Connectivity).** Рішення не використовувати ORM (Object-relational mapping) і натомість скористатися JDBC зумовлене тим, що для належного опрацювання даної теми важливо було забезпечити повне керування SQL-запитами та зверненнями до бази даних, в тому числі необхідна була можливість явного встановлення рівню ізольованості транзакцій, що зручно робити засобами JDBC за допомогою методу **setTransactionIsolation**() інтерфейсу **java.sql.Connection**:

```java
@Override
public void create(UserCredentials user) {
    JdbcDaoConnection daoConnection = new JdbcDaoConnection(connection);
    Connection conn = daoConnection.getConnection();

    try {
        /** Виконуємо операції як одну транзакцію, щоб забезпечити атомарність**/
        daoConnection.begin();

        conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
```

Рисунок 3.2 — встановлення рівню ізольованості транзакції засобами JDBC

3.2. Особливості імплементації

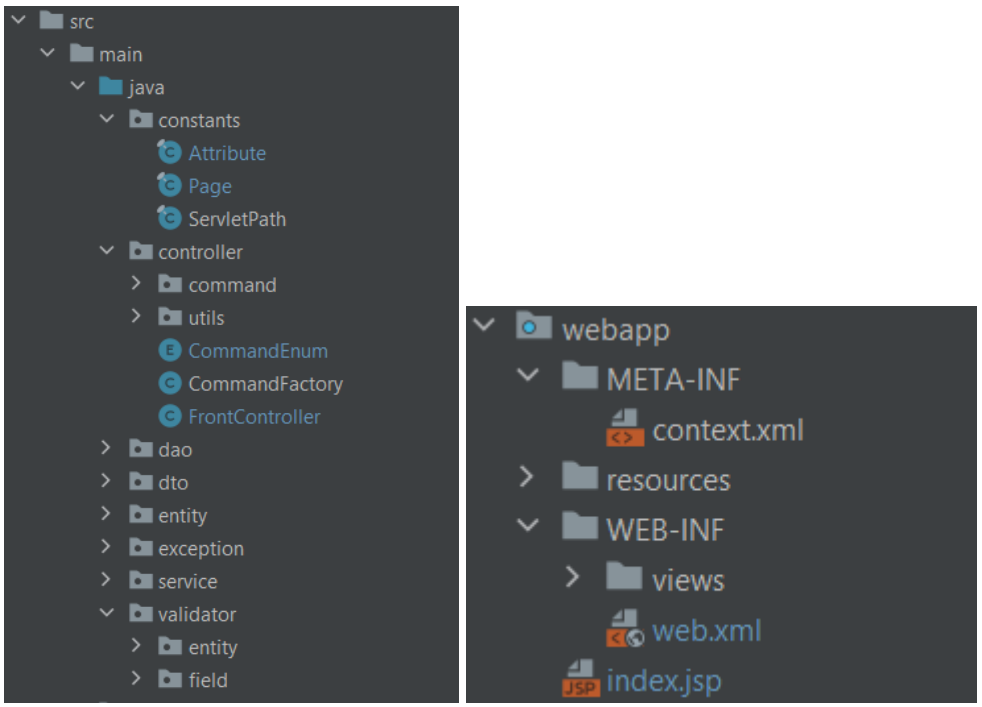

Рисунок 3.3 — структура проекту в середовищі розробки Intellij Idea

Застосунок було імплементовано відповідно до обраного архітектурного шаблону MVC.

Вхідною точкою в застосунок є клас **FrontController** у пакеті controller: завдяки анотації сервлетів на обробку контролеру відправляються всі HTTP-запити, чий URL починається з /chess. Залежно від запиту FrontController викликає відповідну команду з пакету command на його обробку.

```
@WebServlet(urlPatterns = { "/chess/*" }, loadOnStartup = 1)
public class FrontController extends HttpServlet {

    private static final Logger LOGGER = Logger.getLogger(FrontController.class);
    private static final long serialVersionUID = 1L;

    public FrontController() {
    }

    @Override
    public void init() throws ServletException {
        super.init();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
```

Рисунок 3.4 — частина імплементації класу FrontController та його сервлетна анотація

Пакет entity містить Java-класи, що реалізують рівень моделі паттерну MVC; пакет dao реалізує зв’язок з базою даних з використанням JDBC, а пакет service інкапсулює всі можливі операції з базою даних як методи для виклику рівнем контролеру.

Пакети validator, dto, exception та constants містять імплементацію допоміжних функцій: наприклад, під час введення користувачем електронної пошти та паролю попередня перевірка їх з використанням регулярних виразів забезпечується саме валідатором, щоб не робити зайве звернення до бази даних з обліковими даними, які там точно міститись не можуть (наприклад, електронна пошта без символу @).

Зберігання даних про обліковий запис користувача (e-mail, пароль, роль) було вирішено розділити зі зберіганням деталей про нього для того, щоб зменшити навантаження на сервер під час сесії, адже залежно від ролі авторизованого користувача має відрізнятися рівень навігаційної панелі (вкладка “Модерація” доступна лише модераторам та вище). Саме тому user_credentials та user_details є окремими таблицями в БД.

3.3. Демонстрація вирішення конфліктів паралельного доступу в системі

В даній системі рейтинг користувачів може оновлюватись у двох випадках:

1) За результатами рейтингової партії (перемога або програш);
2) Після опрацювання скарги модератором — рейтинг буде повернено, якщо гравця перемогли з використанням підказок; рейтинг буде знижено на 500 одиниць гравцям, яких було помічено за нечесною грою.

Може виникнути ситуація, коли різні транзакції доступаються до даних про рейтинг користувача одночасно — наприклад, якщо користувач виграв рейтингову партію, а одночасно з цим модератор роздивився його скаргу і прийняв рішення про відшкодування одиниць рейтингу в зв’язку з порушенням опонентом правил чесної гри.

Транзакція А	Транзакція Б
BEGIN;	BEGIN;
	UPDATE user_details SET rating_bullet=rating_bullet+10 WHERE user_id='1';
UPDATE user_details SET rating_bullet=rating_bullet+7 WHERE user_id='1';	
COMMIT;	
	COMMIT;

Таблиця 3.1 — Паралельне оновлення рейтингу користувача, що може призвести до аномалій серіалізації

Виникне одна з аномалій серіалізації — втрачене оновлення (**lost update)**, тобто паралельне виконання транзакцій призведе до результату, який неможливо досягнути при будь-якому послідовному виконанні.

Враховуючи імплементацію рівнів ізольованості в PostgreSQL (див. таблицю 1.5), для всіх операцій оновлення рейтингу користувачів в даній АІС було обрано рівень ізольованості Serializable, адже тільки цей рівень дозволяє гарантовано уникати аномалії серіалізації:

```java
@Override
public void updateBulletRating(int userId, int ratingChange) {
    JdbcDaoConnection daoConnection = new JdbcDaoConnection(connection);
    Connection conn = daoConnection.getConnection();

    try {
        daoConnection.begin();
        /**Встановлення рівню ізольованості Serializable для операції з рейтингом**/
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

        try (PreparedStatement updateStatement = conn.prepareStatement(UPDATE_RATING_BULLET)) {
            updateStatement.setInt( parameterIndex: 1, ratingChange);
            updateStatement.setInt( parameterIndex: 2, userId);
            updateStatement.executeUpdate();
        }

        daoConnection.commit();
    } catch (SQLException e) {
        daoConnection.rollback();
        LOGGER.error( message: "Error while updating bullet rating for user ID: " + userId, e);
        throw new ServerException(e);
    } finally {
        if (connectionShouldBeClosed) {
            daoConnection.close();
        }
    }
}
```

Рисунок 3.5 — реалізація операції оновлення рейтингу з використанням рівню ізольованості Serializable, клас JdbcUserDetailsDao

Висновки

Транзакції — фундаментальний механізм для безпечного доступу до даних засобами СКБД. У цій курсовій роботі було проаналізовано питання управління транзакціями в контексті PostgreSQL.

Результати курсової роботи підтверджують важливість грамотного управління транзакціями для забезпечення надійності та цілісності даних в інформаційних системах. Можна зробити висновок, що будь-який розробник програмного забезпечення, яке так чи інакше пов’язане з базами даних, має бути ознайомленим з поняттям транзакції, принципами атомарності, узгодженості, ізольованості та надійності; особливостями різних рівнів ізольованості транзакцій. Недостатня компетентність в цій темі може призвести до невміння вирішувати конфлікти паралельного доступу до даних, що особливо важливо вміти розробникам вебзастосунків.

Вивчені питання було проілюстровано на практиці під час створення вебзастосунку-AIC, що одночасно забезпечує функціонал багатокористувацького шахового додатку і надає адміністраторам можливість керувати базою даних. Додаток було реалізовано мовою Java з використанням технології сервлетів та JSP-сторінок, а доступ до бази даних PostgreSQL в застосунку здійснюється з використанням інтерфейсу JDBC, що надає користувачеві повний контроль над зверненнями до БД мовою SQL та дозволяє явно визначати рівень ізольованості кожної транзакції.

Вивчені концепції та наведені в курсовій роботі наочні приклади можуть бути корисними розробникам додатків, адміністраторам баз даних та всім, хто працює з PostgreSQL або іншими СКБД.

Список використаних джерел

1. Документація PostgreSQL з керування паралельним доступом [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/current/mvcc.html

2. Рівні ізольованості в PostgreSQL [Електронний ресурс] – Режим доступу до ресурсу: https://www.thenile.dev/blog/transaction-isolation-postgres

3. Стандарт SQL-92 [Електронний ресурс] – Режим доступу до ресурсу: https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

4. A Critique of ANSI SQL Isolation Levels [Електронний ресурс] / [H. Berenson, P. Bernstein, J. Gray та ін.]. – 1995. – Режим доступу до ресурсу: https://arxiv.org/pdf/cs/0701157.

5. The Transaction Concept: Virtues and Limitations [Електронний ресурс] / Jim Gray. – 1981. – Режим доступу до ресурсу: http://jimgray.azurewebsites.net/papers/theTransactionConcept.pdf?from=https://research.microsoft.com/~gray/papers/theTransactionConcept.pdf&type=path.

6. Принципи ACID у базах даних [Електронний ресурс] – Режим доступу до ресурсу: https://towardsdatascience.com/database-basics-acid-transactions-bf4d38bd8e26

7. Фази транзакції [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/transaction-states-in-dbms/

8. Рейтинг шахових веб-сайтів [Електронний ресурс] – Режим доступу до ресурсу: https://web.archive.org/web/20140129160838/http://www.alexa.com/topsites/category/Games/Board_Games/Abstract/Battle_Games/Chess

9. Переваги використання MVC [Електронний ресурс] – Режим доступу до ресурсу: https://blog.iandavis.com/2008/12/what-are-the-benefits-of-mvc/

10. Специфікація Java Servlet API [Електронний ресурс] – Режим доступу до ресурсу: https://javadoc.io/doc/javax.servlet/javax.servlet-api/latest/index.html

Додаток А

(обов'язковий)

Перелік прийнятих скорочень

БД — база даних;

СКБД — система керування базами даних;

АІС — автоматизована інформаційна система.

Додаток Б

(обов’язковий)

Реляційна модель даних

user_credentials Відповідає типу сутності “Користувач (облікові дані)”				
Ключ	*Ім’я атрибуту*	*Тип атрибуту*	*Обов’язковість*	*Пояснення (перехід від ER-моделі)*
PK	id	integer (serial)	NN	Простий первинний ключ, автоінкремент.
	nickname	varchar (30)	NN	Простий обов’язковий атрибут “Нікнейм”.
	email	varchar (50)	NN	Простий обов’язковий атрибут “E-mail”.
	password	varchar (50)	NN	Простий обов’язковий атрибут “Пароль”.

	role	varchar (10)	NN	Простий обов’язковий атрибут “Роль”.

user_details Відповідає типу сутності “Користувач (деталі)”				
Ключ	*Ім’я атрибуту*	*Тип атрибуту*	*Обов’язковість*	*Пояснення (перехід від ER-моделі)*
PK, FK	user_id	integer (serial)	NN	Простий первинний ключ; зовнішній ключ, посилається на атрибут id таблиці user_credentials. ON DELETE CASCADE — при видаленні користувача видаляти деталі про нього. ON UPDATE CASCADE — при оновленні id користувача оновлювати його в деталях користувача.

	rating_bullet	integer	NN	Простий обов’язковий атрибут “Рейтинг (буллет)”.
	rating_blitz	integer	NN	Простий обов’язковий атрибут “Рейтинг (бліц)”.
	rating_rapid	integer	NN	Простий обов’язковий атрибут “Рейтинг (рапід)”.
	rating_classic	integer	NN	Простий обов’язковий атрибут “Рейтинг (класика)”.
	country	varchar (255)	N	Простий необов’язковий атрибут “Країна”.
	gender	varchar (10)	N	Простий необов’язковий атрибут “Стать”.
	date_of_birth	date	N	Простий необов’язковий атрибут “Дата народження”.

	profile_picture	varchar (255)	N	Простий необов’язковий атрибут “Фотографія профілю”.
	bio	text	N	Простий необов’язковий атрибут “Опис профілю”.
	account_created_at	timestamptz	NN	Простий обов’язковий атрибут “Дата реєстрації”.

game_request Відповідає типу сутності “Запит на гру”				
Ключ	*Ім’я атрибуту*	*Тип атрибуту*	*Обов’язковість*	*Пояснення (перехід від ER-моделі)*
PK	request_id	integer (serial)	NN	Простий первинний ключ, автоінкремент.

FK	user_id	integer	NN	Зовнішній ключ, посилається на атрибут id таблиці user_credentials. ON DELETE CASCADE — при видаленні користувача видаляти його запити. ON UPDATE CASCADE — при оновленні id користувача оновлювати його в запитах.
	game_type	game_type (enum: “bullet”, “blitz”, “rapid”, “classic”)	NN	Простий обов’язковий атрибут “Тип гри”.
	rating_less	integer	NN	Простий обов’язковий атрибут “Рейтинг супротивника (нижня границя)”.
	rating_more	integer	NN	Простий обов’язковий атрибут “Рейтинг супротивника (верхня границя)”.

	time_restriction	integer	NN	Простий обов’язковий атрибут “Обмеження по часу”.
	preferred_color	varchar (5)	N	Простий необов’язковий атрибут “Бажаний колір”.
	is_rating	boolean	NN	Простий обов’язковий атрибут “Чи є рейтинговою”.

chess_game Відповідає типу сутності “Шахова партія”				
Ключ	*Ім’я атрибуту*	*Тип атрибуту*	*Обов’язковість*	*Пояснення (перехід від ER-моделі)*
PK	game_id	integer (serial)	NN	Простий первинний ключ, автоінкремент.

	time_beginning	timestamptz	NN	Простий обов’язковий атрибут “Час початку”.
	time_end	timestamptz	N	Простий необов’язковий атрибут “Час кінця”.
	game_type	game_type (enum: 'Bullet', 'Blitz', 'Rapid', 'Classic')	NN	Простий обов’язковий атрибут “Тип гри”.
	game_status	game_status (enum: 'Starting' 'Ongoing', 'Finised', 'Cancelled')	NN	Простий обов’язковий атрибут “Статус гри”.
FK	black_user_id	integer	NN	Простий обов’язковий атрибут “Гравець за чорних”.
FK	white_user_id	varchar (5)	NN	Простий обов’язковий атрибут “Гравець за білих”.

FK	winner_id	boolean	N	Простий необов’язковий атрибут “Переможець”.
	moves	varchar(10)[]	N	Простий необов’язковий атрибут “Кроки”.
	is_rating	boolean	NN	Простий обов’язковий атрибут “Чи є рейтинговою”.
	rating_white	integer	NN	Простий обов’язковий атрибут “Рейтинг білих”.
	rating_black	integer	NN	Простий обов’язковий атрибут “Рейтинг чорних”.
	time_restriction	integer	NN	Простий обов’язковий атрибут “Обмеження по часу”.

chat_message

Відповідає типу сутності “Повідомлення чату”				
Ключ	*Ім’я атрибуту*	*Тип атрибуту*	*Обов’язковість*	*Пояснення (перехід від ER-моделі)*
PK	message_id	integer (serial)	NN	Простий первинний ключ, автоінкремент.
FK	sender_id	integer	NN	Зовнішній ключ, посилається на атрибут id таблиці user_credentials. ON DELETE CASCADE — при видаленні користувача видаляти його повідомлення. ON UPDATE CASCADE — при оновленні id користувача оновлювати його в повідомленнях.
FK	game_id	integer	NN	Зовнішній ключ, посилається на атрибут game_id таблиці chess_game.

				ON DELETE CASCADE — при видаленні гри видаляти її чат. ON UPDATE CASCADE — при оновленні id гри оновлювати його в повідомленнях.
	content	text	NN	Простий обов’язковий атрибут “Зміст”.
	timestamp	timestamptz	NN	Простий обов’язковий атрибут “Час відправки”.

complaint Відповідає типу сутності “Скарга”				
Ключ	*Ім’я атрибуту*	*Тип атрибуту*	*Обов’язковість*	*Пояснення (перехід від ER-моделі)*

PK	id	integer (serial)	NN	Простий первинний ключ, автоінкремент.
	created_at	timestamptz	NN	Простий обов’язковий атрибут “Час створення”.
FK	complainant_id	integer	NN	Зовнішній ключ, посилається на атрибут id таблиці user_credentials. ON DELETE CASCADE — при видаленні скаржника видаляти його скарги. ON UPDATE CASCADE — при оновленні id скаржника оновлювати його в скаргах.
FK	reported_id	integer	NN	Зовнішній ключ, посилається на атрибут id таблиці user_credentials. ON DELETE CASCADE — при видаленні оскарженого видаляти скарги на нього.

				ON UPDATE CASCADE — при оновленні id оскарженого оновлювати його відповідно.
	complaint_type	complaint_type (enum: 'Cheat', 'Insult', 'Rating manipulation', 'Trolling', 'Other')	NN	Простий обов’язковий атрибут “Тип скарги”.
	reason	text	N	Простий необов’язковий атрибут “Причина”.
FK	chat_message_id	integer	N	Зовнішній ключ, посилається на атрибут message_id таблиці chat_message. ON DELETE SET NULL — при видаленні повідомлення обнулити посилання. ON UPDATE CASCADE — при оновленні id повідомлення оновлювати посилання.

FK	moderator_id	integer	N	Зовнішній ключ, посилається на атрибут id таблиці user_credentials. ON DELETE SET NULL — при видаленні модератора обнулити посилання. ON UPDATE CASCADE — при оновленні id модератора оновлювати посилання.
	status	complaint_status (enum: 'pending', 'resolved');	NN	Простий обов’язковий атрибут “Статус опрацювання”.