Тестовые функции (Unit Tests)

void testUserRegistrationAndAuthentication();

Тестирует регистрацию и аутентификацию пользователей.

void testAddBookAndAnnotation();

Тестирует добавление книг и работу с аннотациями.

void testSearchBooks();

Тестирует функциональность поиска книг.

void testUserStatusAndBlocking();

Тестирует блокировку/разблокировку пользователей и проверку статуса.

**Тест 1: Регистрация и Аутентификация**

void TestDatabase::testUserRegistrationAndAuthentication()

{

database& db = database::get\_instance(); // Получаем экземпляр БД

// Шаг 1: Успешная регистрация нового пользователя

// QVERIFY2(условие, сообщение\_в\_случае\_ошибки)

QVERIFY2(db.registerUser("testuser1", "Pass123!", "user1@example.com", "client"),

"User 'testuser1' registration should succeed.");

// Шаг 2: Попытка регистрации пользователя с уже существующим логином

// Ожидаем, что эта операция вернет false.

QVERIFY2(!db.registerUser("testuser1", "OtherPass", "user1\_alt@example.com", "client"),

"Registering an existing user 'testuser1' again should fail.");

// Шаг 3: Успешная аутентификация зарегистрированного пользователя

QVERIFY2(db.authenticateUser("testuser1", "Pass123!"),

"Authentication for 'testuser1' with correct credentials should succeed.");

// Шаг 4: Попытка аутентификации с неверным паролем

QVERIFY2(!db.authenticateUser("testuser1", "WrongPass!"),

"Authentication for 'testuser1' with incorrect password should fail.");

// Шаг 5: Попытка аутентификации несуществующего пользователя

QVERIFY2(!db.authenticateUser("nonexistentuser", "Pass123!"),

"Authentication of a non-existent user should fail.");

// Шаг 6: Проверка роли пользователя

// QCOMPARE(фактическое\_значение, ожидаемое\_значение)

QCOMPARE(db.getUserRole("testuser1"), QString("client"));

// Шаг 7: Проверка ID пользователя

int userId = db.getUserID("testuser1");

QVERIFY2(userId > 0, "User ID for 'testuser1' should be a positive integer.");

}

**Тест 2: Статус пользователя и блокировка**

void TestDatabase::testUserStatusAndBlocking()

{

database& db = database::get\_instance();

// Регистрируем пользователя для теста

db.registerUser("blocktestuser", "PassBlock123!", "block@example.com", "client");

int userId = db.getUserID("blocktestuser");

QVERIFY2(userId > 0, "Failed to get ID for 'blocktestuser' for status testing.");

// Шаг 1: Проверка начального статуса (по умолчанию 'active')

QCOMPARE(db.getUserStatus(userId), QString("active"));

// Убедимся, что пользователь может войти

QVERIFY2(db.authenticateUser("blocktestuser", "PassBlock123!"),

"User 'blocktestuser' should be active and authenticatable initially.");

// Шаг 2: Блокировка пользователя

QVERIFY2(db.setUserStatus(userId, "blocked"), "Setting user status to 'blocked' should succeed.");

QCOMPARE(db.getUserStatus(userId), QString("blocked")); // Проверяем, что статус действительно изменился

// Шаг 3: Попытка аутентификации заблокированного пользователя

// Ожидаем, что authenticateUser вернет false для заблокированного пользователя.

QVERIFY2(!db.authenticateUser("blocktestuser", "PassBlock123!"),

"Authentication of a blocked user 'blocktestuser' should fail.");

// Шаг 4: Разблокировка пользователя

QVERIFY2(db.setUserStatus(userId, "active"), "Setting user status back to 'active' should succeed.");

QCOMPARE(db.getUserStatus(userId), QString("active"));

// Шаг 5: Аутентификация разблокированного пользователя

QVERIFY2(db.authenticateUser("blocktestuser", "PassBlock123!"),

"Authentication of a re-activated user 'blocktestuser' should succeed.");

}

**Тест 3: Добавление книги и работа с аннотацией**

void TestDatabase::testAddBookAndAnnotation()

{

database& db = database::get\_instance();

// Шаг 1: Добавление новой книги

QVERIFY2(db.addBookToLibrary("The Test Book", "Anonymous Coder", "2024", "Tech"),

"Adding 'The Test Book' should succeed.");

// Шаг 2: Поиск ID добавленной книги

// Так как addBookToLibrary не возвращает ID, мы должны найти книгу другим способом.

// getAllBooks() возвращает список строк "id,title,author,year,genre,is\_available,annotation".

QStringList books = db.getAllBooks();

int testBookId = -1;

bool foundBook = false;

for (const QString& bookData : books) {

QStringList parts = bookData.split(',');

if (parts.size() > 1 && parts[1] == "The Test Book") { // parts[1] это title

testBookId = parts[0].toInt(); // parts[0] это id

foundBook = true;

break;

}

}

QVERIFY2(foundBook, "The newly added book 'The Test Book' was not found via getAllBooks().");

QVERIFY2(testBookId > 0, "Book ID for 'The Test Book' should be positive.");

// Шаг 3: Обновление аннотации для этой книги

QString annotationText = "This is a detailed annotation for The Test Book. It explains many things.";

QVERIFY2(db.updateBookAnnotation(testBookId, annotationText),

"Updating book annotation for 'The Test Book' should succeed.");

// Шаг 4: Получение и проверка аннотации и названия

// getBookAnnotation(book\_id) возвращает "title:::annotation"

QString titleAndAnnotation = db.getBookAnnotation(testBookId);

QVERIFY2(!titleAndAnnotation.isEmpty(), "getBookAnnotation should not return empty for an existing book.");

QStringList annotationParts = titleAndAnnotation.split(":::");

QCOMPARE(annotationParts.size(), 2); // Ожидаем две части: название и аннотацию

if (annotationParts.size() == 2) {

QCOMPARE(annotationParts[0], QString("The Test Book")); // Проверка правильности названия

QCOMPARE(annotationParts[1], annotationText); // Проверка правильности аннотации

}

// Шаг 5: Попытка получить аннотацию для несуществующей книги

// Используем ID, которого точно нет в базе.

QVERIFY2(db.getBookAnnotation(99999).isEmpty(),

"getBookAnnotation for a non-existent book ID (99999) should return an empty string.");

}

**Тест 4: Поиск книг**

void TestDatabase::testSearchBooks()

{

database& db = database::get\_instance();

// Добавляем несколько книг для тестового поиска

db.addBookToLibrary("Alpha Search Book", "Author A", "2000", "GenreX");

db.addBookToLibrary("Beta Test Title", "Author B", "2001", "GenreY");

db.addBookToLibrary("Gamma Alpha Another", "Author C", "2002", "GenreX");

db.addBookToLibrary("Delta Search Query", "Author A", "2003", "GenreZ");

QStringList searchResults;

// Шаг 1: Поиск по части названия (ожидаем регистронезависимый поиск)

searchResults = db.searchBooks("title", "Alpha");

QCOMPARE(searchResults.size(), 2); // "Alpha Search Book", "Gamma Alpha Another"

// Дополнительная проверка, что найдены нужные книги (можно проверять ID или полные названия)

bool foundAlphaSearch = false;

bool foundGammaAlpha = false;

for(const QString& book : searchResults) {

if (book.contains("Alpha Search Book")) foundAlphaSearch = true;

if (book.contains("Gamma Alpha Another")) foundGammaAlpha = true;

}

QVERIFY(foundAlphaSearch && foundGammaAlpha);

// Шаг 2: Поиск по автору

searchResults = db.searchBooks("author", "Author A");

QCOMPARE(searchResults.size(), 2); // "Alpha Search Book", "Delta Search Query"

// Шаг 3: Поиск по жанру

searchResults = db.searchBooks("genre", "GenreX");

QCOMPARE(searchResults.size(), 2); // "Alpha Search Book", "Gamma Alpha Another"

// Шаг 4: Поиск по полному названию (регистронезависимый)

searchResults = db.searchBooks("title", "beta test title");

QCOMPARE(searchResults.size(), 1);

if (!searchResults.isEmpty()) {

QVERIFY(searchResults.first().contains("Beta Test Title"));

}

// Шаг 5: Поиск, который не должен дать результатов

searchResults = db.searchBooks("title", "NonExistentBookXYZ123");

QVERIFY(searchResults.isEmpty());

// Шаг 6: Поиск с невалидным критерием

// Реализация database::searchBooks возвращает пустой список при неверном критерии.

searchResults = db.searchBooks("invalid\_criteria\_type", "test");

QVERIFY(searchResults.isEmpty());

}

**Как это работает:**

Подробное объяснение запуска описано в видеодокументации.

Подробно о 4-х тестовых функциях:

1. **void TestDatabase::testUserRegistrationAndAuthentication()**
   * **Цель:** Проверить корректность процесса регистрации новых пользователей и их последующей аутентификации. Также проверяется обработка ошибочных сценариев.
   * **Шаги:**
     1. **Успешная регистрация:** Регистрируется пользователь testuser1 с валидными данными. QVERIFY2 проверяет, что db.registerUser() возвращает true.
     2. **Повторная регистрация:** Попытка зарегистрировать пользователя с тем же логином testuser1. QVERIFY2 проверяет, что db.registerUser() возвращает false (так как логин должен быть уникальным).
     3. **Успешная аутентификация:** Попытка войти под testuser1 с правильным паролем. QVERIFY2 проверяет, что db.authenticateUser() возвращает true.
     4. **Неверный пароль:** Попытка войти под testuser1 с неправильным паролем. QVERIFY2 проверяет, что db.authenticateUser() возвращает false.
     5. **Несуществующий пользователь:** Попытка войти под логином, которого нет в базе. QVERIFY2 проверяет, что db.authenticateUser() возвращает false.
     6. **Проверка роли:** QCOMPARE проверяет, что db.getUserRole("testuser1") возвращает ожидаемую роль "client".
     7. **Проверка ID:** QVERIFY2 проверяет, что db.getUserID("testuser1") возвращает положительное число (ID должен быть > 0 после успешной вставки в БД с автоинкрементом).
2. **void TestDatabase::testUserStatusAndBlocking()**
   * **Цель:** Проверить функциональность изменения статуса пользователя (активен/заблокирован) и как это влияет на аутентификацию.
   * **Шаги:**
     1. **Начальный статус:** Регистрируется пользователь blocktestuser. QCOMPARE проверяет, что его начальный статус active. QVERIFY2 подтверждает, что он может аутентифицироваться.
     2. **Блокировка:** Статус пользователя изменяется на blocked с помощью db.setUserStatus(). QVERIFY2 проверяет успешность операции, QCOMPARE – что статус действительно изменился.
     3. **Аутентификация заблокированного:** Попытка аутентификации blocktestuser. QVERIFY2 проверяет, что db.authenticateUser() возвращает false.
     4. **Разблокировка:** Статус пользователя изменяется обратно на active. Проверки аналогичны шагу 2.
     5. **Аутентификация разблокированного:** Попытка аутентификации blocktestuser. QVERIFY2 проверяет, что db.authenticateUser() теперь возвращает true.
3. **void TestDatabase::testAddBookAndAnnotation()**
   * **Цель:** Проверить добавление книг в библиотеку и управление их аннотациями.
   * **Шаги:**
     1. **Добавление книги:** Добавляется книга "The Test Book". QVERIFY2 проверяет, что db.addBookToLibrary() возвращает true.
     2. **Поиск ID:** Так как addBookToLibrary не возвращает ID, книга ищется в списке всех книг (db.getAllBooks()) по названию, чтобы получить ее ID. QVERIFY2 проверяет, что книга найдена и ее ID корректен.
     3. **Обновление аннотации:** Для найденной книги устанавливается аннотация с помощью db.updateBookAnnotation(). QVERIFY2 проверяет успешность.
     4. **Проверка аннотации:** db.getBookAnnotation() вызывается для получения названия и аннотации. QCOMPARE используется для проверки, что и название, и текст аннотации соответствуют ожидаемым.
     5. **Аннотация несуществующей книги:** Попытка получить аннотацию для ID книги, которой нет в базе. QVERIFY2 проверяет, что db.getBookAnnotation() возвращает пустую строку.
4. **void TestDatabase::testSearchBooks()**
   * **Цель:** Проверить корректность работы функции поиска книг по различным критериям.
   * **Шаги:**
     1. **Подготовка:** В базу добавляется несколько книг с различными названиями, авторами и жанрами.
     2. **Поиск по части названия:** Выполняется поиск по слову "Alpha". QCOMPARE проверяет, что найдено ожидаемое количество книг. Дополнительно проверяется, что это именно те книги.
     3. **Поиск по автору:** Поиск по "Author A". QCOMPARE проверяет количество (2).
     4. **Поиск по жанру:** Поиск по "GenreX". QCOMPARE проверяет количество (2).
     5. **Поиск по полному названию (регистронезависимый):** Поиск по "beta test title". QCOMPARE проверяет количество и содержимое.
     6. **Пустой результат:** Поиск по заведомо несуществующему названию. QVERIFY проверяет, что список результатов пуст.
     7. **Невалидный критерий:** Поиск с использованием неверного критерия (например, "invalid\_criteria\_type"). QVERIFY проверяет, что список результатов пуст, так как реализация database::searchBooks должна обрабатывать такой случай.

Каждый из этих тестов спроектирован так, чтобы быть независимым и проверять небольшой, конкретный участок функциональности класса database.