Полезные SQL-запросы для тестировщика

Краткий чек-лист запросов к базе данных API Практикума

🔒 База изолирована: ты видишь только записи, созданные твоим API-ключом. Не пугайся, если таблицы выглядят полупустыми.

1. Проверка создания записи

```
SELECT * FROM users
ORDER BY id DESC
LIMIT 5;
```

Зачем: проверяет, что отправленный через POST /v1/api/users кандидат действительно записался в базу, и его поля (имя, возраст, статус) сохранились без искажений.

Когда:

- Сразу после POST-запросов — подтвердить успешную вставку.
- Проверить, что id увеличивается корректно.

2. Простая фильтрация данных

```
-- Фильтр по конкретному статусу
SELECT * FROM users
WHERE status = 'candidate';

-- Фильтр по возрасту
SELECT * FROM users
WHERE age >= 18;
```

Зачем: быстро получить срез данных из базы и сверить его с ответом API (например, с результатом GET /v1/api/users?status=candidate).

Когда:

- Сверка работы фильтров: API вернул 10 кандидатов, а база выдаёт 12 — ты нашёл баг фильтрации.
- Ручной поиск тестовых данных для других тестов.

3. Авто-аудит бизнес-логики

```
SELECT id, name, age, status FROM users
WHERE (age < 18 AND status != 'minor')
   OR (age BETWEEN 18 AND 65 AND status != 'candidate')
   OR (age >= 66 AND status != 'retired');
```

Зачем: один запрос проверяет всю таблицу на соответствие ТЗ. Вернулась хотя бы одна строка — ты нашёл баг в логике назначения статусов.

Когда:

- Проверка автоматического назначения статуса по возрасту.
- За секунду протестировать всю базу после нагрузочного теста.

4. Поиск дубликатов (лишние пропуски)

```
SELECT user_id, COUNT(*) FROM screening_passes
GROUP BY user_id
HAVING COUNT(*) > 1;
```

Зачем: по ТЗ на каждого кандидата выдаётся ровно один скрининговый пропуск. Если у одного user_id несколько пропусков — это баг. Кстати, в v1 метод POST /users как раз создаёт 2 пропуска вместо 1 — этот запрос поможет его поймать.

Когда:

- Сразу после POST /v1/api/users — проверить, сколько пропусков создалось.
- Поиск «двойных» записей при тестировании выдачи пропусков.

5. Сквозная проверка связей (JOIN)

```
SELECT u.name, u.status, p.pass_code, p.status AS pass_status
FROM users u
JOIN screening_passes p ON u.id = p.user_id;
```

Зачем: проверяет побочные эффекты: при создании пользователя ему должен автоматически выдаваться пропуск. JOIN связывает пользователя и его пропуск — вся картина на одном экране.

Когда:

- Интеграционные сценарии: создался ли пропуск, тот ли у него user_id.

6. Поиск зависших статусов (просрочка)

```
SELECT id, pass_code, expires_at, status
FROM screening_passes
WHERE expires_at < NOW() AND status = 'active';
```

Зачем: если срок пропуска истёк (expires_at < NOW()), но статус всё ещё 'active' — это логический баг: либо не отработала фоновая очистка (Maintenance), либо сломалась логика обновления статуса.

Когда:

- Тестирование жизненного цикла пропусков и фоновых задач очистки.

7. Аудит статистики

```
SELECT status, COUNT(*) FROM users
GROUP BY status;
```

Зачем: быстрая сверка статистики: совпадают ли цифры на экране с реальными данными в базе.

Когда:

- Тестирование дашбордов, счётчиков и аналитики.

🚀 База данных — твой главный свидетель

БД никогда не врёт, в отличие от ответов API. Ответ может вернуть 200 OK, но только прямой запрос в базу покажет, сохранились ли данные правильно. Сверяй каждый запрос к API с состоянием базы!