ChatGPT

Взаимодействие клиент–сервер (API) и основы баз данных

Часть 1: Взаимодействие клиент–сервер – основы API и форматы данных

Что такое API?

API (Application Programming Interface) – это «интерфейс прикладного программирования», то есть описание способов, как одни программы могут взаимодействовать с другими и обмениваться данными [1]. Проще говоря, API задаёт **набор правил и методов**, по которым клиент (например, веб-приложение) может попросить у сервера выполнить какое-то действие или предоставить данные, и получить определённый ответ. API можно представить как **контракт** между клиентом и сервером: если клиент формирует запрос по правилам API, сервер гарантирует понятный ответ.

Аналогия: API похож на **меню в ресторане**. Клиент выступает посетителем, сервер – кухней, а API – официантом, который передаёт заказы на кухню и приносит результаты [2]. Вам не нужно знать, **как именно** кухня готовит блюдо или считает счёт – достаточно сделать заказ по меню (то есть в соответствии с правилами API), и официант-API вернёт готовое блюдо или сообщение. Все сложные детали скрыты; клиент и кухня общаются через понятный интерфейс (официанта). Это упрощает взаимодействие: программы могут сотрудничать, не зная внутреннего устройства друг друга.

HTTP-запросы и ответы

Схема: клиент отправляет HTTP-запрос серверу, сервер обрабатывает его и возвращает HTTP-ответ клиенту.
В веб-разработке наиболее распространён API для клиент–серверного взаимодействия – это **Web API** на основе протокола **HTTP**. Общение происходит через HTTP-запросы и ответы:
1. **Клиент отправляет запрос** на сервер (например, запросить определённые данные или выполнить действие).
2. **Сервер обрабатывает запрос** – выполняет необходимую логику, возможно, обращается к базе данных или другим сервисам.
3. **Сервер возвращает ответ** клиенту – обычно это запрошенные данные или результат операции [3].

Каждое HTTP-сообщение (и запрос, и ответ) имеет стандартную **структуру** [4]:
- **Стартовая строка** – для запроса она содержит метод (тип действия) и URL ресурса, для ответа – код состояния и статус (например, `HTTP/1.1 200 OK`).
- **Заголовки (Headers)** – пары ключ:значение с дополнительной информацией: тип контента, параметры кэширования, авторизация и т.д. Например, в запросе может быть заголовок `Content-Type: application/json` (указание типа данных в теле).
- **Тело (Body)** – необязательная часть. В запросе тело обычно присутствует у методов,

передающих данные (например, JSON при POST-запросе). В ответе тело содержит сами данные, которые сервер возвращает (HTML-страницу, JSON, файл и т.п.).

Пример структуры запроса и ответа: клиент может отправить запрос:

```
GET /api/users HTTP/1.1
Host: example.com
Accept: application/json
```

Это означает: метод GET, ресурс `/api/users`, HTTP версия 1.1, ожидаем JSON-данные. Если всё успешно, сервер вернёт ответ:

```
HTTP/1.1 200 OK
Content-Type: application/json

[ ... данные ... ]
```

Где код 200 OK означает успех, а в теле будет JSON с запрошенными пользователями.

Основные методы HTTP

Протокол HTTP поддерживает несколько методов (типы запросов). В контексте веб-API методы принято трактовать как **глаголы действий** над ресурсами [5]:

- **GET** – получить ресурс (только чтение данных). *Например:* `GET /users` – получить список всех пользователей. (Безопасный метод, не меняет данные на сервере.)
- **POST** – создать новый ресурс или передать данные. *Например:* `POST /users` – создать нового пользователя, передав его данные в теле запроса.
- **PUT** – полное обновление существующего ресурса. *Например:* `PUT /users/123` с телом запроса обновит **все поля** пользователя с ID 123.
- **PATCH** – частичное обновление ресурса. *Например:* `PATCH /users/123` с телом, содержащим только изменяемые поля, обновит **часть данных** пользователя 123.
- **DELETE** – удаление ресурса. *Например:* `DELETE /users/123` – удалить пользователя с ID 123.

Эти пять – самые распространённые методы в RESTful API. Их можно ассоциировать с операциями **CRUD** (Create, Read, Update, Delete). Важно: HTTP-метод в запросе указывает серверу намерение: например, GET – только получить данные, POST – добавить новые [6]. Сервер должен корректно реагировать: обычно, запрос на удаление возвращает код успеха только если ресурс действительно был удалён и т.д. Также есть другие методы (HEAD, OPTIONS и др.), но в базовом тестировании API они встречаются реже.

Форматы данных: JSON и XML

Обмениваясь данными, клиент и сервер должны «говорить на одном языке» в теле запроса/ответа. На сегодня **JSON (JavaScript Object Notation)** – наиболее популярный формат сериализации данных в веб-API [7]. JSON представляет собой текст в виде пар «ключ: значение», очень похожий на синтаксис JavaScript-объектов. Например:

```
{
  "name": "Alice",
  "age": 25,
  "isStudent": false
}
```

Преимущество JSON – он **легко читается человеком** и парсится программой. Строки и имена полей записываются в двойных кавычках, числа и логические значения – без кавычек. Структуры данных (объекты, массивы) могут вкладываться друг в друга. Формально JSON – просто текст, но его структура позволяет однозначно представить сложные данные. В отличие от JavaScript-объектов, в JSON имена свойств **обязательно в двойных кавычках**, недопустимо использовать одинарные или без кавычек [8]. Это делает JSON независимым от конкретного языка программирования.

Еще один распространённый ранее формат – **XML (Extensible Markup Language)**, где данные хранятся в виде тегов, напоминающих HTML. Пример того же объекта в XML:

```
<user>
  <name>Alice</name>
  <age>25</age>
  <isStudent>false</isStudent>
</user>
```

XML более громоздок, и современные веб-API используют его реже, хотя в некоторых системах (например, SOAP-сервисы) он ещё встречается [9]. **JSON вытеснил XML** в новых API благодаря простоте и компактности. Для полноты можно упомянуть также форматы YAML, ProtoBuf, CSV – но для тестировщика начального уровня главное уверенно понимать JSON, так как большинство API возвращают именно его.

Пример вызова API на практике

Чтобы увидеть API в действии, вызовем публичный сервис, который возвращает данные в JSON. Например, есть открытое API для определения возраста по имени – **Agify.io**. Если в браузере открыть URL:

```
https://api.agify.io?name=Michael
```

браузер отобразит примерно такое содержимое:

```
{"name":"Michael","age":54,"count": (число)}
```

Здесь `name` – принятое имя, `age` – предполагаемый возраст, `count` – сколько раз имя встречалось в выборке. Браузер не стилизует JSON, а просто показывает «сырые» данные. **Причина:** мы обратились прямо к API-серверу, минуя красивый интерфейс – сервер вернул JSON, и браузер отобразил его как текст. Это демонстрирует, что фронтенд (например, скрипт на странице) мог бы получить эти данные и использовать их.

Пример: фрагмент JSON-данных, возвращённых API (список задач из тестового сервиса).
На изображении показан ответ от тестового API: это массив объектов JSON, каждый из которых представляет задачу (id, title, completed). Такие данные легко понять человеку и обработать программно. В реальном приложении JavaScript-код на странице может сделать такой же запрос и обновить интерфейс (например, показать список задач) без перезагрузки страницы.

Инструменты для тестирования API: Postman и DevTools

Для тестировщика важно уметь самостоятельно вызывать API вне самого приложения. Один из самых популярных инструментов – **Postman**. Postman – это приложение, позволяющее вручную формировать HTTP-запросы: выбрать метод (GET, POST и т.д.), указать URL, заголовки, тело запроса (например, JSON). Нажав «Send», вы получите ответ от сервера: код состояния, заголовки и тело ответа.

В Postman удобно тестировать разные сценарии: проверять ответы на корректные и некорректные запросы, экспериментировать с параметрами. Он отображает ответ (JSON) в удобном формате с подсветкой, показывает время отклика, размер и пр. Например, тестировщик может отправить через Postman запрос `POST /users` с JSON телом `{"name": "Test"}` и убедиться, что сервер вернёт код **201 Created** и в ответе новый объект пользователя с присвоенным ID.

Помимо Postman, есть и другие инструменты: **Insomnia**, **SoapUI**, расширения браузера, или даже просто командная строка с `curl`. Но Postman стал индустриальным стандартом благодаря удобству и функциональности (настройка коллекций запросов, автоматизация, тест-скрипты и др.). В нашем курсе мы уже ознакомились с основами Postman; сейчас можно углубить знания: например, научиться использовать **коллекции и окружения** (для группировки запросов и параметров), или писать простые **тесты** на JavaScript прямо в Postman (проверять, что поле в ответе имеет нужное значение). Эти навыки помогут в реальной работе, хотя для начала достаточно и базового умения отправлять запросы.

Еще один полезный инструмент – **встроенные средства разработчика браузера (Developer Console)**. Во вкладке **Network** браузера можно наблюдать, какие запросы отправляет фронтенд приложения к серверу, и какие ответы приходят. Тестировщик может: - Проверять, что при нажатии кнопки на сайте ушёл правильный запрос (метод, URL, параметры) и сервер вернул ожидаемые данные.
- Смотреть детали каждого запроса/ответа: заголовки, коды, тело ответа JSON.
- Использовать консоль браузера для отладки – например, выполнить в консоли `fetch()` запрос к API и посмотреть результат прямо там, либо отследить ошибки (например, CORS-проблемы или 404).

Таким образом, DevTools помогают **увидеть взаимодействие фронтенда с API в реальном времени**, а Postman – имитировать такие запросы вручную независимо от интерфейса. Оба подхода важны: сначала можно отработать ответы API в Postman, а затем убедиться, что реальное приложение корректно с этим API общается.

Практика: самостоятельные запросы в Postman

Чтобы закрепить материал, желательно практиковаться. Хорошая новость: есть публичные тестовые API, где можно ничего не сломать! Например, **JSONPlaceholder** – это фейковый онлайн-API, предоставляющий ресурсы типичного приложения (пользователи, посты, задачи и т.д.). Попробуйте в Postman отправить запрос:

```
GET https://jsonplaceholder.typicode.com/todos/1
```

Он должен вернуть JSON объекта *“Todo”* (задачи) с id=1. Убедитесь, что вы получили ответ **200 OK** и тело с полями `userId`, `id`, `title`, `completed`. Это означает, что запрос прошёл успешно, ресурс найден. Можно поэкспериментировать: изменить ID на несуществующий (например 99999) – тогда сервер JSONPlaceholder вернёт пустой объект или код **404 Not Found**, показывая, что ресурса нет.

Практическое задание для тренировки: через Postman выполнить POST-запрос на `https://jsonplaceholder.typicode.com/posts` с каким-нибудь JSON в теле (например, `{"title": "Test", "body": "Hello", "userId": 5}`). Этот сервис не создаёт реальных данных, но должен вернуть вам в ответ сгенерированный объект с новым `id`. Так вы увидите, как сервер подтверждает создание ресурса кодом **201 Created** и отдаёт детали созданного объекта.

Не забудьте, что при тестировании API важно **внимательно проверять**: - **Код ответа** (200, 404, 500 etc.) соответствует ожидаемому сценарию, - **Тело ответа** содержит правильные данные, - **Побочные эффекты** (например, реально ли добавилась новая запись — хотя на тестовом API это не сохраняется).

Эти навыки вы отточите на практике, выполняя разные запросы и анализируя ответы.

Коды ответов HTTP

HTTP-ответ сервера всегда содержит **код статуса** – трёхзначное число, разделённое на категории по первой цифре:
- **2xx Успех:** 200 OK (успешный запрос, стандартный код по умолчанию) [10] ; 201 Created (успешно создан ресурс, обычно ответ на POST); 204 No Content (успех, но без тела ответа, например при успешном DELETE).
- **3xx Перенаправление:** 301 Moved Permanently, 302 Found (частичный или временный редирект), 304 Not Modified (можно использовать кешированный ответ). В тестировании API реже актуальны, но знать полезно.
- **4xx Ошибка клиента:** 400 Bad Request (неправильный запрос, например, отсутствует обязательный параметр или неверный формат данных); 401 Unauthorized (необходима авторизация – обычно вы не предоставили нужный токен/API-ключ); 403 Forbidden (доступ запрещён – у вас нет прав на этот ресурс); 404 Not Found (ресурс не найден по данному URL). Тестировщик проверяет, что **некорректные запросы возвращают корректные ошибки**: например, запрос с неверным ID должен вернуть 404, а не 200 с пустыми данными.
- **5xx Ошибка сервера:** 500 Internal Server Error (общая ошибка на сервере, например, из-за необработанного исключения); 502 Bad Gateway, 503 Service Unavailable (сервер недоступен или перегружен), 504 Gateway Timeout (таймаут при проксировании запроса). В идеале, таких ошибок не должно быть, но если бэкенд падает, тестировщик фиксирует код 500 и описание проблемы.

Важно: При тестировании API обращайте внимание, что сервер возвращает **уместный код**. Неправильно, если, скажем, при ошибке валидации сервера вы получили 200 OK с сообщением об ошибке в теле – правильно вернуть 4xx. Коды – часть контракта API с клиентом. Хорошо задокументированный API всегда указывает, какие коды и в каких ситуациях возможны.

Безопасность API: ключи и доступ

В открытых примерах мы вызывали API, доступные всем. Но множество реальных веб-API требуют **аутентификации** – подтверждения, что вы имеете право ими пользоваться. Часто для этого применяются **API-ключи** или токены доступа.

API-ключ – это уникальная строка (набор символов), как пароль, выдаваемый разработчиком сервиса пользователю или приложению [11]. Ключ обычно передаётся в каждом запросе (например, в заголовке `Authorization: Bearer <YOUR_API_KEY>` или как параметр `?api_key=...`). Сервер знает ваш ключ и даёт доступ только если ключ верный и не превышены лимиты. Примеры: чтобы обратиться к API погоды (например, OpenWeatherMap) или к сервисам Google, нужно зарегистрироваться и получить ключ. Если запрос отправлен без ключа или с неверным, сервер вернёт **401 Unauthorized** – отказано в доступе.

Для тестировщика важно **уметь работать с защищёнными API**: - Хранить ключи в секрете (не светить их в общедоступных местах, не коммитить в открытые репозитории). - Знать, куда вставлять ключ (в заголовок, в параметры или в теле – зависит от документации API). - Проверять, что система правильно реагирует на отсутствие/неверный ключ (т.е. выдаёт 401/403).

Кроме API-ключей, есть и другие методы аутентификации: **OAuth-токены**, cookie с сессией, JWT-токены. Но суть одна: клиент должен предъявить «удостоверение». При тестировании API вы можете получить тестовый ключ или токен от разработчиков и использовать его в Postman (хорошая практика – хранить такие вещи в **переменных окружения** Postman, чтобы не копировать вручную в каждый запрос).

Наконец, отметим аспект **SSL/TLS**: практически все современные API доступны по HTTPS (шифрованному протоколу). В адресе вы видите `https://` – это значит, данные запроса и ответа передаются в зашифрованном виде. Как тестировщик, вы редко столкнётесь с нюансами шифрования (этим занимаются админы), но всегда используйте правильный URL (https://, не http://) согласно документации API, иначе может быть блокировка запросов браузером из соображений безопасности.

Закрепление материала и связь с предыдущими темами

Мы рассмотрели, как фронтенд и бэкенд общаются через API. Коротко подчеркнём главное: - **API** – это понятный интерфейс для запросов и ответов между программами. Для клиента веб-сервиса API обычно представлено набором URL-эндпойнтов и правил их вызова. - **HTTP** – базовый протокол веба; методы HTTP аналогичны действиям (получить, создать, удалить...), коды ответов информируют о результате. - **JSON** – основной формат данных в веб-API сегодня, который легко читается и используется практически во всех современных проектах. Его знать критически важно (да, есть XML и др., но их применяют реже). - **Postman** – друг тестировщика: с его помощью можно вручную проверять API независимо от фронтенда. Это особенно полезно, когда UI приложения ещё сырой или нужно изолированно проверить логику сервера. - **API без UI:** Тестировщики нередко проверяют бэкенд **напрямую через API**, чтобы убедиться, что внутренняя логика работает правильно, ещё до того, как появится красивый интерфейс. Например, можно через API создать “левых” данных или вызвать скрытый сценарий, который сложно воспроизвести через UI. - **Связь с фронтендом:** Хотя мы не углублялись в JavaScript, вы должны понимать: браузерный JS может делать HTTP-запросы (через `fetch`, AJAX и т.д.) к API даже после загрузки страницы. Это называется **AJAX-запросы**. Именно так современные страницы динамически обновляются. Например, лента соцсети подгружает новые посты через API без перезагрузки всей страницы. Для тестировщика это значит: некоторые баги могут быть на стыке фронта и API – например,

фронт неправильно обработал ответ от сервера. Поэтому нужно знать, где искать проблему: в интерфейсе (Developer Tools) или на стороне API (посмотреть реальный ответ в Postman). - **Документация API:** Хорошие команды документируют свои API – будь то в формате Swagger/OpenAPI (интерактивная HTML-страница со списком эндпойнтов), либо просто Wiki-страница с описанием. Тестировщик должен уметь читать такую документацию. Если API задокументировано через Swagger, можно даже отправлять тестовые запросы прямо из браузера, что упрощает ознакомление с возможностями сервиса.

Вопросы для самопроверки: *Что такое API своими словами? В чём отличие метода GET от POST? Почему JSON столь популярен сегодня?* Если вы можете ответить, значит, материал усвоен. В реальном проекте, имея эти знания, вы будете уверенно общаться с разработчиками (понимая их термины “эндпойнт”, “HTTP-метод”, “боди запроса”, “JSON”), и тестировать не только видимые функции, но и невидимый обмен данными между фронтом и бэком.

Часть 2: Основы баз данных – работа с данными на сервере

Что такое база данных?

Современные веб-приложения должны где-то хранить информацию – будь то список пользователей, товары в магазине, посты блога или результаты матчей. **База данных (БД)** – это специализированное хранилище данных и система управления этим хранилищем. В принципе, данные можно хранить и в файлах, но при большом объёме и сложных связях это неудобно и ненадёжно. База данных обеспечивает **структурированное хранение и быстрый доступ** к данным, а также их целостность и безопасность. Проще говоря, БД – это сердце бэкенда: когда вы регистрируетесь на сайте или публикуете что-то, информация сохраняется в базе; когда вы входите или просматриваете контент – данные считываются из базы.

Существует несколько типов баз данных. Самый распространённый класс – **реляционные базы данных (SQL-базы)**. Они получили такое название, потому что представляют данные в виде таблиц и могут устанавливать **отношения (relations)** между таблицами. Альтернативы – **NoSQL-базы**, о них скажем ниже, но начнём с реляционной модели, т.к. она повсеместно используется и наверняка будет в ваших проектах.

Реляционная модель: таблицы, строки, столбцы

Реляционная база данных хранит данные в **таблицах**, очень похожих на лист Excel. Таблица состоит из **столбцов** (колонок) и **строк** (записей). Каждый столбец определённого типа (число, текст, дата и т.д.) описывает одно свойство, а каждая строка содержит значения этих свойств для одной сущности. Например, рассмотрим таблицу **Users (Пользователи)**: она может иметь столбцы `id`, `name` (имя), `email`, `age` (возраст). Тогда одна строка таблицы Users представляет одного пользователя с конкретными значениями этих полей. Формально, таблица – это **набор связанных данных, организованных по столбцам и строкам** [12].

Например, таблица **Users** может выглядеть так:

id	name	email	age
1	Alice	alice@example.com	30
2	Bob	bob@example.com	17

id	name	email	age
3	Charlie	charlie@example.com	25

Здесь:
- Каждый **столбец** имеет название и тип данных (id – число, name – текст, и т.д.). Он содержит значения этого типа для **каждой строки** [13]. Например, столбец age для всех записей хранит числа (возраст).
- Каждая **строка** – отдельная запись, объединяющая все поля. Строка 1 это пользователь Alice, 30 лет, с таким-то email. Строка 2 – Bob, 17 лет, и т.п.

Реляционная модель означает, что можно устанавливать **связи между таблицами**. Например, можно иметь таблицу Orders (Заказы) с колонкой user_id – в ней хранится id пользователя, сделавшего заказ. Это user_id ссылается на id в таблице Users. Таким образом, Order «связан» с конкретным пользователем. Связи обычно выражаются через соответствие значений (внешние ключи). В нашем примере каждый заказ с user_id=1 принадлежит Alice (id=1 в Users). Реляционные СУБД поддерживают целостность связей – нельзя, например, добавить заказ с несуществующим user_id (если настроены ограничения целостности).

Итого: база данных может состоять из множества таблиц, каждая хранит данные об определённых объектах (пользователи, заказы, продукты и т.д.), а связи между ними позволяют собирать комплексную информацию по запросу. Например, **JOIN-запрос** может объединить данные из Users и Orders, чтобы выбрать все заказы с именами пользователей.

Язык SQL и основные команды

Для работы с реляционными БД используется специальный язык – **SQL (Structured Query Language)**, “язык структурированных запросов”. Почти все СУБД (MySQL, PostgreSQL, Oracle, MS SQL, SQLite и др.) понимают SQL (с незначительными вариациями). С помощью SQL можно делать две основные вещи: 1. **DDL (Data Definition Language)** – определять структуру базы: создавать/изменять/удалять таблицы, индексы, пользователи и права и т.п. (Команды: CREATE, ALTER, DROP...). 2. **DML (Data Manipulation Language)** – манипулировать самими данными: добавлять, извлекать, обновлять, удалять записи. Эти нас интересуют особенно, ведь тестировщик часто читает и проверяет данные. К DML-командам относятся **SELECT, INSERT, UPDATE, DELETE**.

Рассмотрим основные команды DML на простых примерах. Пусть у нас есть таблица **Students** со столбцами: Name, BirthYear, Grade (имя, год рождения, оценка).

- **SELECT** – выборка данных (чтение). Это самая частая операция: получить данные, соответствующие условию. Простейший пример:

```
SELECT Name, Grade
FROM Students
WHERE Grade >= 90;
```

Этот запрос выберет имена и оценки всех студентов, у которых `Grade >= 90`. Разберём: после SELECT перечисляются нужные столбцы (здесь два), FROM указывает таблицу (Students), опционально WHERE задаёт условие фильтрации (только записи, удовлетворяющие условию). Результатом будет **таблица-набор**, содержащая строки

студентов-отличников с указанными полями. Если убрать WHERE, получили бы всех студентов. Можно выбирать все столбцы, используя `SELECT *` (звёздочка означает «все поля»).

- **INSERT** – добавление новой записи. Пример:

```
INSERT INTO Students (Name, BirthYear, Grade)
VALUES ('Иван', 2005, 85);
```

Этот запрос добавит новую строку в таблицу Students с Name = 'Иван', BirthYear = 2005, Grade = 85. Синтаксис: после `INSERT INTO` идёт название таблицы и список столбцов, которые мы заполняем, а после VALUES – соответствующие значения в скобках. Если запрос выполнится успешно, в БД появится новая запись. (Обычно СУБД вернёт информацию о добавленных строках, а если настроен автоинкремент id – то и новый id.)

- **UPDATE** – обновление существующих записей. Пример:

```
UPDATE Students
SET Grade = 95
WHERE Name = 'Иван';
```

Здесь мы говорим: в таблице Students найти строки, где Name = 'Иван', и для них установить новое значение Grade = 95. Команда UPDATE требует указать таблицу, затем оператор SET задаёт, какие поля поменять и на что, а WHERE (очень важно!) ограничивает, какие строки обновлять. Без WHERE обновятся **все** записи, поэтому условие всегда нужно, если вы не намерены изменить всё. В нашем примере после выполнения запроса у студента Иван оценка станет 95 вместо 85.

- **DELETE** – удаление записей. Пример:

```
DELETE FROM Students
WHERE Name = 'Иван';
```

Удалит все записи, где Name = 'Иван'. В нашем случае – ту самую запись Ивана. Опять же, **без условия WHERE удалятся все строки в таблице**, поэтому всегда осторожно.

Общие моменты: - SQL не чувствителен к регистру команд (SELECT и select эквивалентны), но имена таблиц/столбцов могут быть чувствительны (в зависимости от СУБД). Обычно ключевые слова пишут прописными для наглядности. - Кавычки: строки в SQL берутся в одинарные кавычки ('Иван'), а имена идентификаторов могут обрамляться в двойные или квадратные – но чаще пишут без кавычек, если имя без пробелов и спецсимволов. - Результат SELECT – это **таблица** (может быть ноль, одна или много строк). В тестировании вы часто будете выполнять SELECT, чтобы **проверить данные**. Например, после шага «пользователь обновил профиль» – можно SELECT'ом прочитать запись этого пользователя и убедиться, что изменения сохранились верно. - INSERT/UPDATE/DELETE изменяют данные. Обычно после них тестировщик делает SELECT, чтобы убедиться, что запись добавилась/изменилась/удалилась. В некоторых случаях, если есть доступ, можно использовать **транзакцию** (BEGIN/ROLLBACK) чтобы потом откатить изменения – но это

продвинутый трюк, на начальном этапе можно работать на тестовой базе, где не жалко изменять данные.

Практика для тренировки SQL: попробуйте на бумаге (или мысленно) составить запросы для таких задач: *«Найдите всех пользователей старше 18 лет»* (SELECT с условием `WHERE age > 18`); *«Добавить нового пользователя с именем Катя, email kate@xyz, age 22»* (INSERT … VALUES ('Катя','kate@xyz',22)); *«Повысьте всем студентам оценки на 5, у кого сейчас меньше 60»* (UPDATE с `SET Grade = Grade + 5 WHERE Grade < 60`). Если вы понимаете, как составить эти запросы, то у вас уже есть базовые навыки работы с данными в БД.

Взаимодействие сервера с БД

Вернёмся к архитектуре веб-приложения: у нас есть клиент (браузер) и сервер (бэкенд). База данных обычно находится либо на том же сервере, либо на отдельном сервере, но **клиент (браузер) не обращается к БД напрямую** – все запросы идут через бэкенд. Когда фронтенд хочет получить или изменить данные, он вызывает API сервера, а сервер уже формирует нужные SQL-запросы к базе.

Пример: пользователь логинится на сайте. Что происходит на бэкенде? Код сервера примет запрос (например, POST `/login` с логином и паролем). Затем сервер выполнит к базе примерно такой SQL:

```
SELECT * FROM Users
WHERE username = '<логин_пользователя>'
  AND password_hash = '<хеш_пароля>';
```

Он пытается найти запись пользователя с совпадающим логином и паролем. Если SELECT ничего не вернул – сервер отправит ответ, что логин/пароль неверны (401 Unauthorized). Если вернул – сервер создаст сессию/токен и пришлёт успех. То есть **бизнес-логика сервера завязана на операциях с БД**. Другой пример: при оформлении заказа интернет-магазин: бэкенд добавит запись в таблицу Orders (INSERT), спишет товар со склада (UPDATE таблицы Products, уменьшив поле stock), создаст транзакцию платежа и т.д.

Почему важно знать тестировщику: даже если вы не пишете сами SQL в коде, понимание, что под капотом бэкенд делает с базой, помогает писать лучше тест-кейсы. Например, сценарий: «пользователь удалил аккаунт, данные должны пропасть из системы» – вы можете проверить не только отсутствие информации в UI, но и убедиться через БД, что запись пользователя удалена из таблицы Users, его заказы помечены как отключенные и т.п. То же с **граничными случаями**: если вводимое значение слишком большое, возможно, вы получите ошибку базы (например, строка превышает длину колонки). Понимая устройство таблиц, можно предвидеть такие ошибки и проверить их.

Важно: Приложения часто используют специальные библиотеки – **ORM (Object-Relational Mapping)**, которые позволяют разработчику работать с базой не напрямую через SQL, а через кодовые объекты. Например, вместо писать `SELECT * FROM Users WHERE id=5`, программист пишет что-то вроде `User.find(5)` или вызывает метод `user.save()`. ORM сама генерирует SQL. Для тестировщика это не меняет сути: в итоге **SQL-запросы всё равно выполняются**, и можно наблюдать их эффекты. Но стоит понимать, что не всегда вам дадут прямой доступ к базе

на проекте – иногда проверки придётся делать косвенно через API. Однако в учебных задачах и тестовых средах доступ обычно есть, и нужно уметь им пользоваться.

Транзакции и надёжность данных

Представьте ситуацию: банковское приложение переводит деньги с аккаунта А на аккаунт Б. В базе это две операции: вычесть сумму со счёта А, добавить к счёту Б. Что если после первой операции что-то пойдёт не так (сбой сети, ошибка)? Нельзя допустить, чтобы деньги снялись с А и не дошли до Б – данные будут неконсистентны.

Для решения таких проблем существуют **транзакции**. Транзакция объединяет несколько изменений в одно целое: либо выполнится **всё целиком**, либо, при любой ошибке, **ничего не выполнится**. Это называют принципом "все или ничего" – гарантия **атомарности** операций [14]. В примере с переводом денег: операции списания и зачисления выполняются в рамках одной транзакции – если вторая часть не удалась, первая будет отменена автоматически. Транзакции обеспечивают также **согласованность** (данные переходят из одного целостного состояния в другое), **изоляцию** (параллельные транзакции не мешают друг другу) и **долговечность** (после коммита данные гарантированно сохранены) – вместе эти свойства известны как ACID [15] [16].

Для нас важно знать: транзакции – это механизм надёжности в СУБД. В тестировании вы, возможно, столкнётесь с ними, когда будете проверять сценарии, например: *«Если в процессе регистрации шаг 3 провалился, данные из шагов 1-2 не должны сохраниться».* Это именно транзакционный подход: либо регистрация полностью успешна, либо всё откатывается. Если вы видите частично сохранённые данные – это баг.

В практическом плане, когда вы работаете напрямую с БД (через клиент типа DBeaver), будьте аккуратны: по умолчанию многие СУБД либо **автоматически выполняют** каждый ваш запрос (auto-commit), либо позволяют вам начать транзакцию. Не делайте в боевой базе ничего, что вы не уверены как откатить. В тестовых же базах можно тренироваться. Иногда в тесте можно использовать транзакцию: начать, выполнить нужные изменения (например, подготовить данные), а после сценария сделать ROLLBACK, чтобы вернуть базу в исходное состояние.

Инструменты для работы с БД

Работать с базой напрямую – значит выполнять SQL-запросы и просматривать таблицы. Для этого есть множество инструментов. Популярные **клиентские программы для БД**: - **DBeaver** (универсальный кросс-СУБД клиент, бесплатный) [17] – позволяет подключаться к разным базам (MySQL, PostgreSQL, Oracle, etc.), выполнять запросы, экспортировать данные. - **MySQL Workbench** (для MySQL/MariaDB), **pgAdmin** (для PostgreSQL) – фирменные GUI-интерфейсы для соответствующих СУБД. - Веб-интерфейсы: **phpMyAdmin**, **Adminer** – веб-приложения для управления базой (часто ставятся на локальный сервер, например, вместе с XAMPP для MySQL). - **SQLiteStudio/Browser** – если база на SQLite (файловая), есть простые утилиты.

Все они предоставляют похожие возможности: **просмотр схемы** (списка таблиц, столбцов), просмотр содержимого таблицы (первые N строк или отфильтровано), редактор SQL-запросов с подсветкой синтаксиса, возможно, визуальное построение запросов. Пользоваться ими довольно интуитивно: выбираете нужное соединение/таблицу, нажимаете «Select top 100 rows» или вводите вручную запрос.

Для тестировщика основной сценарий – **сделать SELECT запрос** к базе, чтобы убедиться в наличии/отсутствии/правильности данных. Например, после регистрации нового пользователя –

вы можете в клиенте БД выполнить `SELECT * FROM Users WHERE email='тот_что_ввели'` и проверить, что запись появилась с корректными данными (имя, роль по умолчанию, статус активен и т.д.). Если нужно протестировать удаление – можно до действия зафиксировать count*() записей, выполнить действие, потом снова count()* и убедиться, что количество уменьшилось на 1.

Другой сценарий – **подготовка данных через БД**. Иногда быстрее вставить в базу тестовые данные напрямую (особенно, если UI не предоставляет нужного способа). Например, чтобы протестировать отображение 1000 товаров на странице, вы можете сгенерировать и выполнить INSERTы для быстрого наполнения таблицы фейковыми товарами, вместо того чтобы через интерфейс добавлять по одному. Но тут важно координироваться с командой, чтобы не нарушить целостность приложения.

Подытожим: умение подключиться к базе и выполнить пару запросов – необходимый навык тестировщика. В некоторых компаниях на интервью даже дают простое задание по SQL. Это не значит, что вы обязаны быть экспертом, но базовые операции (SELECT/INSERT/UPDATE/DELETE с WHERE) должны быть в вашем арсенале.

Практика: составление простых SQL-запросов

(Эта секция рассчитана больше на учебную аудиторию.) Чтобы научиться, нужны упражнения. Вот несколько заданий, попробуйте сначала сами сформулировать SQL, а ниже сверитесь с примером ответа:

1. **Выбрать всех пользователей старше 18 лет из таблицы Users.**
   Решение: `SELECT * FROM Users WHERE age > 18;` – вернёт все столбцы (`*`) тех записей, где возраст больше 18.

2. **Найти email пользователей с именем "John".**
   Решение: `SELECT email FROM Users WHERE name = 'John';` – список email’ов всех Джонов.

3. **Посчитать, сколько заказов в таблице Orders сделал пользователь с id=5.**
   Решение: `SELECT COUNT(*) FROM Orders WHERE user_id = 5;` – вернёт число заказов (функция агрегирования COUNT).

4. **Добавить новую категорию товаров "Garden" в таблицу Categories.**
   Решение: `INSERT INTO Categories (name) VALUES ('Garden');`

5. **Повысить зарплату всем сотрудникам на 10%, у которых производительность > 8.** (Предположим, таблица Employees(col: salary, performance)).
   Решение: `UPDATE Employees SET salary = salary * 1.10 WHERE performance > 8;`

6. **Удалить все временные записи из таблицы Sessions, старше 2023-01-01.**
   Решение: `DELETE FROM Sessions WHERE created_at < '2023-01-01';`

Потренировавшись на таких примерах, вы постепенно начнёте «думать на SQL». В реальности, конечно, запросы могут быть сложнее (с JOIN, подзапросами, группировками), но большинство проверок для тестирования связаны с относительно простыми условиями.

Другие виды хранилищ: NoSQL и современные подходы

Мы говорили о реляционных (SQL) базах, однако следует знать, что мир баз данных шире. **NoSQL** – условное название нереляционных баз данных, которые хранят данные не в таблицах. Их несколько типов: - **Документоориентированные БД** (например, MongoDB): хранят данные в виде документов (обычно формат JSON или близкий). По сути, вместо таблицы Users – **коллекция документов**, где каждый документ – это JSON-объект произвольной структуры (поля могут отличаться от документа к документу) [18] [19]. Это даёт гибкость: можно легко добавлять новые поля. Минус – сложнее делать сложные выборки, нет привычных связей, вместо SQL – свои механизмы запросов. Пример MongoDB-документа для пользователя:

```
{
  "_id": ObjectId("..."),
  "name": "Alice",
  "age": 30,
  "emails": ["alice@example.com", "alice@work.com"]
}
```

Здесь `_id` – уникальный идентификатор (генерируется как ObjectId), остальное – как JSON. MongoDB позволяет хранить вложенные структуры, массивы прямо в документе. - **Ключ-значение хранилища** (например, Redis): представьте гигантский словарь, где по уникальному ключу можно быстро получить значение. Ключи и значения могут быть строками, числами, сериализованными объектами. Такие БД очень быстрые, используются для кеширования, сессий, счетчиков. Но сложные структуры данных на них не построить без дополнительных усилий. - **Графовые БД** (Neo4j): хранят данные в узлах и рёбрах графа – удобно для задач, где важны отношения между объектами (социальные сети, пути в навигации). - **Поисковые движки** (Elasticsearch): специализированные хранилища, оптимизированные под полнотекстовый поиск, аналитические запросы.

Зачем все это знать тестировщику? На базовом уровне – чтобы иметь представление, что не все данные живут в таблицах. Однако большинство приложений, с которыми вы будете работать, всё же используют реляционную БД для основной информации (пользователи, транзакции и т.п.). NoSQL часто применяется как дополнение: например, хранить сессии в Redis, логи в Elasticsearch, кэши в MongoDB и т.п. Если проект использует такую технологию, вам, конечно, расскажут и научат ею пользоваться. Но **принципы работы с данными** остаются: есть некий запрос – есть результат, просто не SQL, а другой синтаксис (пример: запрос в MongoDB пишется в JSON-стиле).

На этапе обучения важно твёрдо освоить реляционные базы, так как понимание таблиц и связей – фундамент для любых структурированных данных.

Зачем тестировщику знать SQL?

Резюмируем с акцентом на роль тестировщика. Понимание базы данных и умение выполнить запрос дают ряд преимуществ: - **Глубокая проверка данных.** Вы не ограничены интерфейсом приложения. Вы можете залезть "под капот" и посмотреть, что реально творится в базе. Бывают случаи, когда на UI всё выглядит нормально, а в базе – грязь (например, дубли, неправильные флаги) и это потом вылезает багами. Тестировщик, который проверяет и базу, ловит такие проблемы раньше. - **Валидация операций.** Пример: при отмене заказа приложение показало “Заказ отменён”. Вы можете проверить в БД: статус этого заказа стал “canceled”, записи об оплате корректно помечены, связанные товары вернулись на склад. Если что-то из этого не произошло –

значит, баг на бэкенде (UI ведь уже не показывает заказ, а он в базе активен – несоответствие!). - **Подготовка тестовых данных.** Как упоминалось, иногда быстрее и надёжнее вставить/править данные в базе вручную при подготовке сложных сценариев. Конечно, без фанатизма и всегда согласуя с разработчиками, если база общая. Но навык экономит время. - **Автоматизация и SQL.** Даже если вы не пишете сложные автотесты сейчас, многие инструменты позволяют делать запросы к базе в рамках тестов. Например, UI-тест может после действий сходить в базу и проверить, что в таблице появилось ожидаемое. Или API-тест проверит, что после вызова POST /create, запись есть в БД. Знание SQL здесь прямая польза. - **Общение с разработчиками и аналитиками.** В команде вам могут давать подсказки типа: "проверь, а пришёл ли такой-то объект, вот SQL" – и скинуть запрос. Вы должны понять и выполнить. Или вы можете сами сообщить баг, подкрепив его результатом SQL-запроса (например: *"Замечено, что после удаления пользователя его данные остаются в таблице X (скриншот результата SELECT...). Ожидается, что запись тоже удалится."*). Такие детали показывают ваш профессионализм.

Отметим, практически все курсы для тестировщиков включают основы SQL. В реальной работе практически наверняка придётся его использовать. Как говорится, *"one does not simply"* быть тестировщиком, не умея сделать SELECT . Серьёзно, этот навык настолько стандартный, что многие работодатели считают его обязательным для QA. Вот цитата: *«Для более точного тестирования базы данных тестировщик должен очень хорошо знать команды SQL и DML»* [20] . Не обязательно "очень хорошо" в смысле писать сложнейшие запросы, но CRUD-операции и простые JOINы – да.

Вывод: клиент–сервер–база как единое целое

Теперь у нас сложилась общая картина, как фронтенд, бэкенд и база данных взаимодействуют: - **Клиент** (браузер с HTML/CSS/JS) отправляет запросы к **серверу** (по HTTP). - **Сервер** (бэкенд-приложение) выполняет бизнес-логику: обрабатывает запросы, и при необходимости обращается к **базе данных** (по SQL) чтобы получить или записать информацию. Затем сервер отправляет ответ клиенту. - **База данных** хранит всю нужную информации и обеспечивает целостность и доступ к ней.

Для качественного тестирования веб-приложения надо учитывать все три слоя. Мы постепенно освоили основы каждого: от верстки и JS на фронте, до API и работы с данными на бэке. На следующем этапе нашего обучения мы перейдём к тому, как тестировать всё это богатство: будем учиться составлять тест-кейсы, работать с системами контроля версий, continuous integration, и другими инструментами командной разработки. Но крепкое понимание технических основ (таких как HTTP и SQL) выгодно отличает хорошего тестировщика. Вы сможете не просто следовать чек-листам, а понимать причину проблем и эффективнее общаться с разработчиками.

Поздравляем с освоением материала! Пользуйтесь полученными знаниями на практике: экспериментируйте с запросами в Postman и SQL-утилитах. Это повысит вашу уверенность в работе с реальными проектами. Успехов в дальнейших шагах в тестировании!

---

1 2 7 9 API: Простое объяснение и аналогия с рестораном
https://icoder.uz/programmirovanie/fxa_2rllzti-what-is-an-api/

3 Как работают HTTP-запросы
https://selectel.ru/blog/http-request/

4 5 6 Типы HTTP-запросов и философия REST / Хабр
https://habr.com/ru/articles/50147/

8 JSON: методы, ограничения, примеры использования.
https://purpleschool.ru/blog/all_about_JSON

10 REST API: Как общаются программы в интернете | Блог о маркетинге
https://www.qmedia.by/blog/rest_api-_kak_obshhayutsya_programmy_v_internete.html

11 Что такое API-ключ? – Подробнее об API-ключах и токенах – AWS
https://aws.amazon.com/ru/what-is/api-key/

12 Таблица (база данных) - Википедия
https://ru.wikipedia.org/wiki/
%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_(%D0%B1%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%

13 Столбец (база данных) - Википедия
https://ru.wikipedia.org/wiki/
%D0%A1%D1%82%D0%BE%D0%BB%D0%B1%D0%B5%D1%86_(%D0%B1%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%

14 16 17 20 Полное руководство по тестированию баз данных
https://qarocks.ru/database-testing-process/

15 Что такое база данных SQL? - Подробно о базах данных SQL - AWS
https://aws.amazon.com/ru/what-is/sql-database/

18 19 MongoDB и PostgreSQL — разница между базами данных — AWS
https://aws.amazon.com/ru/compare/the-difference-between-mongodb-and-postgresql/