Шпаргалка

Архитектура приложения

Архитектура приложения — это способ организации компонентов приложения. В веб-приложении в архитектуру могут входить фронтенд, бэкенд, API и база данных.

API — это интерфейс, который помогает приложениям взаимодействовать. Через него они обмениваются данными посредством запросов и ответов.

API как часть бэкенда «пишет» разработчик — обычно по спецификации или техническому заданию.

База данных (БД) — это набор данных, которые хранятся в электронном виде. Доступ к ним пользователь получает через компьютер.

Базы данных устроены по-разному. Чаще всего встречаются **реляционные** — когда информация хранится в таблицах.

С БД взаимодействует бэкенд: он отправляет ей запросы и получает данные в ответ.

Когда тестируешь API, важно проверить, что изменения — добавление данных, изменение или удаление — корректно записались в базу.

Когда ты знаешь, как устроена архитектура приложения, можно:

- генерировать больше вариантов тестирования, потому что ты разбираешься, как работают компоненты приложения;

- быстрее локализовать баг в любом компоненте;
- более точно локализовать баг, чтобы сэкономить время разработчика на устранение.

Архитектуру приложения можно выстраивать по-разному. Самые распространённые способы — по SOAP и REST.

От архитектурного стиля API зависит, как компоненты архитектуры обмениваются сообщениями: по какому протоколу и в каком формате. От этого зависит, как именно и какими инструментами ты будешь тестировать приложение.

REST — один из способов спроектировать приложение. Его главная особенность — простота отображения данных. В других архитектурных стилях данные «обёрнуты» в дополнительный слой разметки. В REST дополнительного слоя нет: по запросам и ответам сразу понятно, что происходит с данными.

Протокол HTTP

Клиент отправляет серверу **HTTP-запрос,** а сервер возвращает **HTTP-ответ**.

Метод сообщает серверу, какое действие нужно выполнить. Самые распространённые методы:

- GET — запрашивает данные;
- POST — создаёт или получает данные;
- DELETE — удаляет данные;
- PUT — обновляет данные.

URL — адрес, по которому клиент запрашивает данные, а сервер передаёт.

Код состояния (Status Code) сообщает, успешно ли сервер обработал запрос. Например, код 200 означает, что всё прошло успешно. А код 500 указывает на

внутреннюю ошибку сервера.

Текст состояния — текст, который сопровождает код состояния. Например, ошибка 500 — «внутренняя ошибка сервера».

JSON

JSON (JavaScript Object Notation) — это «лёгкий» формат данных, который часто используется при обмене данными между приложениями. Основное преимущество — JSON легко писать и читать. На JSON можно структурно описать практически любой объект, чтобы использовать в системе.

Перед тобой пары вида ключ:значение. «Ключ» — это имя параметра объекта. Например, фамилия, отчество или имя. «Значение» указывается после двоеточия.

```
{
  "first_name": "Ivan",
  "last_name": "Ivanov",
  "middle_name": "Ivanovich",
  "age": 35,
  "is_married": true,
  "has_kids": false,
  "emails": ["ivanov.i.i.@yandex-work.ru", "ivan@yandex-home.ru"],
  "favorite_numbers": [0, 7, 42],
  "pets": [{"type": "dog", "name": "Шарик"}, {"type": "cat", "name": "Мурлыка"}]
}
```

Параметры перечисляют через запятую и описывают в определённом формате. Например, ключ всегда берут в двойные кавычки; значение — в зависимости от типа данных. Текстовые данные заключают в кавычки.

Обрати внимание: набор пар заключён в фигурные скобки — так ты можешь указать, что они относятся к одному и тому же объекту.

Числа не включают в кавычки.

В «Семейном положении» действует условная бинарная логика:

- `true` — правда
- `false` — ложь

Запись значения в квадратных скобках указывает на список, или массив. Он может содержать одно значение — `["type":"dog"]`, несколько — `["type":"dog", "type":"cat"]` или быть пустым. Например, если у человека нет почтовых адресов, но данные об этом всё равно передаются. Это будет выглядеть так: `"emails": []`.

Внутри массива данные просто перечисляют через запятую, и порядок имеет значение. Для сравнения: внутри объектов, где параметры определяются парами `ключ:значение`, порядок может быть любой.

Объект может быть пустым. Так же, как и массив. В этом случае значение будет записано просто фигурными скобками без содержимого между ними: `{}`. Если нужно явно указать, что параметру не присвоено никакое значение, применяют запись вида `"pets": null` — слово `null` говорит, что значение этого параметра не определено.

Postman

Postman — инструмент для тестирования API. Он помогает отправлять HTTP- и HTTPS-запросы в API и получать ответы.

В нём ты можешь:

- выбрать метод запроса;
- добавить и отредактировать URL, по которому нужно отправить запрос;
- задать параметры запроса;
- добавить запросу заголовки;
- составить тело запроса.

В Postman есть функциональность Environment («Окружение»). Она позволяет настроить переменную (Variable) и поставить её в разные запросы.

Если поменяешь значение переменной в одном месте, оно поменяется везде.

Это поможет тебе не менять URL вручную. Вместо адреса ты поставишь переменную — и достаточно один раз поменять её значение, чтобы оно подтянулось во все запросы.

cURL

cURL — приложение, которое позволяет «общаться» с серверами прямо из консоли. Ты пишешь текстовую команду, и в API отправляется запрос.

Команду cURL удобно отправлять коллегам: тестировщикам, разработчикам, аналитикам, менеджерам. Её легко скопировать и переслать в любом мессенджере, а ещё можно добавить в требования, чек-лист, тест-кейс, баг-репорт.

Запрос из Postman можно преобразовать в команду cURL.

Команду cURL можно преобразовать в запрос Postman.

Баг-репорт по API

Обязательные поля **баг-репорта по API** немного иные:

- В шагах нужно обязательно написать **метод запроса**, **параметры**, **URL** и **тело запроса** — если оно есть.
- В окружении нужно указать адрес сервера, на который отправляется запрос.
- В ОР должен быть ответ, описанный в документации или на её основе.
- В ФР должен быть ответ, который был получен по факту.

Дополнительную информацию и остальные необязательные параметры указывают по необходимости.

В баг-репорт можно добавить:

- данные, которые пришлось добавить в базу данных для тестирования;
- ссылку на документацию API — или хотя бы привести цитату из неё;
- обоснование, почему это баг и какие у него могут быть последствия. Например, API стало возвращать дополнительный статус. API, которое стоит рядом, тоже его получает и «падает»: это приводит к ошибкам.

Логи — это записи о событиях, которые происходят в приложении.

События в API тоже логируются. В логах API обычно указывают:

- HTTP-запрос с описанием параметров, заголовков, тела;

- ошибки, которые возникли в ходе обработки запроса.

Логи можно скопировать и приложить в баг-репорт.

Кроме Postman, логи могут записываться в отдельный текстовый файл или в специальные базы данных.

Документация API

Документация API описывает, что умеет делать программный интерфейс приложения: какие запросы может принимать и какие ответы отправлять. Это отдельная документация: она отличается от документации к бэкенду.

Документация помогает разобраться в работе API определённого компонента или приложения. Внутри описаны:

- Действия, которые можно делать с системой; данные, которые можно запросить.
- Ограничения передаваемых данных. Например, у некоторых параметров может быть ограничение на тип данных (число или строка), на длину (от 2 до 50 символов).
- Структура запросов, ответов и адресов, по которым можно отправлять запросы.

Чтобы упростить работу с документацией, пользуются инструментами, которые генерируют её автоматически. Например, инструмент **Swagger**.

Часть сервиса, к которой отправляет запрос клиент, называют **эндпоинт** (endpoint, «конечный пункт»). Эндпоинт также называют «метод API», «URL ресурса» или просто «ручка».

Через документацию в **Swagger** можно сразу отправлять запросы, как в Postman. Для этого нужно выбрать эндпоинт и нажать кнопку Try it out.

Есть и другой инструмент работы с документацией — **Apidoc**.

У Swagger и Apidoc похожая механика работы: документация API автоматически генерируется из файла. Они отличаются дизайном интерфейса и техническими особенностями генераторов, которые не заметны пользователю.

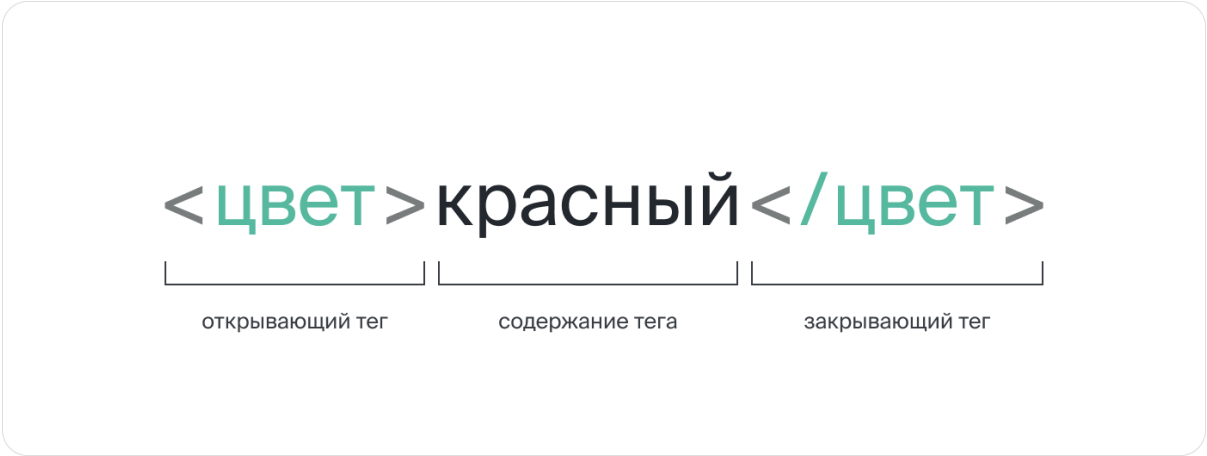
У Swagger больше функциональных возможностей. Например, тебе уже удалось отправить запрос сразу из документации в Swagger, чтобы проверить функцию; в Apidoc так сделать нельзя.

XML

XML — ещё один способ размечать данные, которыми обмениваются приложения.

Документ XML состоит из **элементов** — как из кирпичиков**.** Элемент состоит из **тегов** и **содержимого**.

Тег в начале называется **открывающим,** а в конце — **закрывающим**. Оба тега обязательные.



Элементы в XML можно группировать. Например, «город», «улица» и «дом» можно объединить в «адрес»: так нагляднее.

```
<адрес>
  <город>Москва</город>
  <улица>Первомайская</улица>
  <дом>5</дом>
</адрес>
```

Элемент-контейнер для остальных называется **родителем,** а элементы внутри — **потомками**. «Адрес» — родитель, «город», «улица» и «дом» — потомки.

Родитель одновременно может быть и потомком. Например, «адрес» может лежать внутри элемента «школа»:

```
<школа>
  <адрес>
    <город>Москва</город>
    <улица>Первомайская</улица>
    <дом>5</дом>
```

```
  </адрес>
</школа>
```

Иногда получается так, что в документе есть несколько групп с одинаковыми тегами. Например, в документе `объекты_строительства` есть блоки `<школа>` и `<жилой_дом>`. Оба содержат `<адрес>`.

Если имена тегов одинаковые, а их содержимое разное по структуре, программа не сможет прочитать элементы. Например, адрес школы содержит только текстовые данные, а адрес дома — контейнер с дочерним элементом.

Чтобы программа не запуталась, используют **неймспейс** (англ. namespace). Это указание, что элементы принадлежат одной сущности: например, все вложенные элементы в адресе — дому, а не школе.

XSD (XML Schema Definition) — это описание структуры XML-документа. Иногда разработчики составляют XSD ещё до того, как писать XML.

Документ XSD описывает:

- из каких родительских и дочерних элементов состоит XML;
- как называются теги;
- какие у тегов атрибуты;
- другие детали структуры.

Иногда нужно проверить, соответствует ли XML-документ схеме XSD. Если XML написан не по XSD-схеме, программа не сможет его прочитать.

SOAP

SOAP (Simple Object Access Protocol) — **протокол обмена данными**. Это означает, что клиент и сервер «общаются» по определённым правилам. Так они точно знают, что друг от друга ожидать: в каком формате придут данные и как прислать ответ. Получается меньше сбоев и ошибок.

У протокола есть и недостаток: нужно передавать больше данных, а скорость обработки запросов снижается.

В SOAP API:

- сервер принимает и обрабатывает сообщения строго по спецификации SOAP;
- на URL нужно отправлять сообщения в формате XML. Сервер не сможет прочитать данные в других форматах;
- протоколы передачи данных — HTTP и HTTPS. Можно использовать и другие: например, SMTP — протокол передачи электронной почты.

Если API функциональности разработано по SOAP, она называется **SOAP-сервисом**.

Чтобы составить SOAP-запрос, нужно:

1. написать XML: составить структуру и указать данные, которые будут передаваться в теле SOAP-запроса;
2. упаковать XML в SOAP-запрос.

WSDL

WSDL — это **язык описания веб-сервисов** (Web Services Description Language).

Веб-сервис — это приложение с API, которое доступно по сети: ему можно отправить запрос и получить ответ.

WSDL описывает правила взаимодействия веб-сервисов:

- как нужно составить запрос серверу;
- какие типы данных можно использовать;
- в каком формате должен вернуться ответ.

Благодаря WSDL тестировщик может проверить работу веб-сервиса, который использует SOAP:

- вызвать методы и убедиться, что они выполняют то, что задумано;
- посмотреть, правильно ли передаются аргументы;
- проверить описание типов.

Обычно это делают через инструменты тестирования SOAP — например, SOAP UI. Тестировщик загружает в него WSDL-документ, а сервис генерирует SOAP-сообщение и помогает сравнить полученный результат с ожидаемым.

Проектирование тестов

Чтобы спроектировать тест с **классами эквивалентности**, нужно:

1. Составить запрос, который проверяет, что пользователя действительно можно создать.
2. Проверить, что запрос работает.
3. Выделить классы эквивалентности.
4. Использовать запрос как шаблон: подставить в него несколько вариантов данных — по классам эквивалентности.

Чтобы спроектировать тест с **граничными значениям**, нужно проделать всё то же самое, что и в классах эквивалентности:

- составляешь позитивный запрос и проверяешь, что он работает;
- используешь запрос как шаблон и подставляешь в него подготовленные значения параметра;
- отправляешь запросы и сравниваешь фактические результаты с ожидаемыми.

Валидация

Валидация — механизм, который проверяет, что запрос в API составили правильно: указали данные в нужном формате и не ошиблись в структуре.

Правила валидации разрабочики прописывают в коде. Это делают как на стороне клиента, так и на стороне сервера. Лучше «зашивать» валидацию на бэкенде: на клиенте её может обойти злоумышленник, а к серверу у него обычно нет доступа.

Валидация может предупреждать о том, что:

- **структура запроса неправильная**: например, нет всех нужных параметров или тегов;
- **данные записаны некорректно**: слишком много символов или не тот формат.

Что может пойти не так со структурой запроса:

- **неправильный тип данных в теле**: например, число вместо строки или наоборот;

- **неправильная структура тела запроса**: например, указан формат XML вместо JSON;
- **XML в теле запроса без необходимых тегов.**

Валидация может сработать, когда в запросе пришли данные, недопустимые по требованиям. Если ввести недопустимые по требованиям данные — например, поставить знаки препинания или спецсимволы — сработает валидация.