Выбор протокола передачи данных

Привет, и речь тут пойдёт про разные протоколы передачи данных между системами и сервисами. Конечно же в Backend'е (и между бэкендом и фронтендом). Эта статья - продолжение цикла статей об архитектурах и паттернах (часть 1, часть 2). Ну, кому интересно, добро пожаловать к прочтению.

Хотя не совсем правильно их называть *протоколами* - ведь не все из них являются *протоколами*. Тут более корректный термин - это "*спецификация*", "*стек*" или "*стандарт API*", но для простоты понимания будем всех их называть "*протоколами*". Начнём с основного - глобально, все протоколы передачи данных можно отнести на четыре глобальные группы (типы):

- **REST** (**Resource-Oriented API**) - когда все сущности - это ресурсы, которые управляются стандартными **HTTP** методами
- **RPC** (**Remote Procedure Call**) - это вызов удалённых методов/процедур на стороннем сервисе
- **Гибридные подходы** - протоколы, которые в разной степени совмещают две предыдущих группы
- **Событийные** - как правило, это *Event-Driven* подходы, а построены на **AMQP** или **Apache Kafka**

Так же все эти протоколы можно разделить по способу транспорта:

- **HTTP/HTTPS** - протоколы, которые общаются по **HTTP**, причём поддерживают **HTTP** от версии **1.1** (который имеет поддержку из коробки почти везде)
- **HTTP/2** - протоколы, которые построенны на базе **HTTP 2**, и без клиента, который поддерживает **HTTP 2** не заработают
- **TCP/UDP** - протоколы, которые работают на базе **TCP** или **UPD**. Но чаще всего мы будем сталкиваться с **TCP**
- **AMQP**, **MQTT**, **Kafka** - ну тут всё понятно - это *Event-Driven* системы, **Message Brokers**

Да, все спецификации можно группировать ещё по многим параметрам, но мы будем идти по первому списку типов, поясняя, что бы выбрать, для чего и так далее. Ну и конечно, будем описывать особенности, какой формат общения используется и все вот такие подобные вещи. Начнём.

REST (Resource-Oriented API)

Это тот *архитектурный стиль* для **API** (не протокол), с которым мы сталкиваемся чаще всего. В 2000-м году Рой Филдинг описал набор *принципов и ограничений* для взаимодействия между компонентами системы, и чаще всего - между **клиентом и**

сервером. Основной принцип очень прост - всё представлено как **ресурс**, и каждый ресурс имеет свой уникальный **URL**. Все операции с ресурсами выполняются через стандартные **HTTP методы**:

- **GET** - *чтение*, получить ресурс или список ресурсов
- **POST** - *создание*, создать ресурс
- **PUT**/**PATCH** - *обновление*, обновить ресурс (**PUT** - полная замена, **PATCH** - частичное)
- **DELETE** - *удаление*, удалить ресурс

Так же характеризуется этот принцип ещё и тем, что каждый запрос должен быть **Stateless** (без состояния) - всё, о чем нужно знать серверу должно передаваться в этом запросе, например токены авторизации и прочее. Сервер не должен хранить состояние между запросами, кроме авторизационных токенов или куков. Ответы не имеют строгого контракта, но должны использовать стандартные **HTTP** коды состояния ответа для классификации, например `200 OK` для успешного ответа, `404 Not Found` если ресурс не найден, или `500 Internal Server Error` в случае внутренней ошибки сервера. Со списком всех кодов можно ознакомиться тут - все их перечислять нету смысла, потому что их очень много. В качестве формата данных может использоваться **JSON** или **XML** (да, **REST API** - это не обязательно **JSON**), ну а то вдруг кто-то из вас не знал (да, вордпрессники или битриксоиды?). Стандарт вообще в форматах никак не ограничивает, можно хоть через **CSV** общаться. Эта та спецификация, с которой и мы, бэкендеры, и фронтендеры, и iOS (и даже под нищие ведроид, хотя парадокс - хоть ведроид и нищий, но разрабы на нём получают как под iOS, а иногда и больше) разработчики сталкиваемся чаще всего. **REST** очень простой, каких-то замудрённых и сложных правил нету, писать его можно вообще на чём угодно практически (я даже пробовал **REST** сервис в **DDD** парадигме на языке D-Lang поднимать и там оно работало отлично. Какой каламбур - *DDD on D*), есть куча инструментов, библиотек и фреймворков под классический **REST** (например ныне почивший *Lumen by Laravel*, или *Spiral*/*Slim*), целый стандарт для документации (**OpenAPI**) и визуализации (*Stoplight*, *Redoc*, *SwaggerUI*). Но этому подходу также присущи ряд недостатков - нету строгой схемы контрактов, и всё общение можно формировать "как попало", он не эффективен для сложных связанных данных (overfetching/underfetching), и иногда он становится "слишком простым" для сложных бизнес-процессов. Но во всех остальных случаях - очень хороший стиль, и всё, что не требует очень специфичной логики можно реализовывать через **REST**.

RPC (Remote Procedure Call)

Концепция и класс протоколов **Remote Procedure Call** (**RPC**) - такая концепция, где клиент вызывает *удалённую функцию* или метод на *удалённом сервере* так, как будто бы это была обычная локальная функция. Вся эта концепция строится вокруг вызова *сторонних процедур*, а **не** работы с *ресурсами как в REST*. Основная идея - клиент *отправляет вызов* "*метода*" с определёнными *параметрами на сервер*, *сервер* в

свою очередь *исполняет эту функцию*, и возвращает либо результат, либо ошибку. Вот и всё. По большей части мы не завязываемся на **HTTP** методы и **HTTP** коды, а следуем правилам конкретного **RPC** протокола/стандарта, даже если протокол - **RPC over HTTP** (как например **JSON/XML-RPC**). Основные отличия от **REST**:

- Вызов "*процедур*" ("*действий*" типа `sendEmail` или `calculateDiscount`), а не управление "*объектами*" и "*ресурсами*" через стандартные методы
- Чаще всего у **RPC** единая точка входа, например URL `/rpc`, внутри которого и идёт вызов "*процедуры*"
- В большинстве случаев - строгие контракты, строгая схема и автогенерация серверов/клиентов (кроме **JSON-RPC**, но и там схема довольно явная)
- Поддержка *стриминга* или *bidirectional взаимодействия* (**gRPC**) - в **REST** такого нет, и нужно будет "городить" отдельный **WebSocket**-сервер

RPC чаще всего используется в межсерверных вызовах, либо же в вызовах "*мобайл клиент - сервер*", а для классического фронтенда используются либо **REST**, либо **GraphQL** (но бывают исключения в виде **JSON-RPC**). Всё дело в том, что сама концепция **RPC** может быть слишком избыточна для классических веб-приложений, ибо чаще всего мы оперируем *ресурсами*, и нам гораздо легче манипулировать ресурсами через **HTTP** методы (`GET /users` - получить юзера, `POST /users` - создать юзера), чем делать вызовы `getUser`, `createUser` и тому подобное. Ну и ещё большинство **RPC**-протоколов - *либо бинарные*, *либо имеют строгие контракты* и бывает сложно маппить структуры на фронтенде. Но эти протоколы лучше всего использовать там, где **REST** - уже мало, или требуется *высокая скорость* (*микросервисы* и Highload), или требуются сложные *бизнес-процессы* (это про "*REST - уже мало*"). А для *browser-based* фронтенда взаимодействия может быть избыточно. Ну и ещё момент - браузеры не умеют работать с бинарными протоколами или строгими контрактами, которые как раз и используются в большинстве **RPC** протоколов. "*Яркие представители*" **RPC** - **gRPC**, **SOAP**, **JSON-RPC**, **XML-RPC**, и о них мы поговорим чуть позже.

Гибридные подходы

Здесь мы останавливаться не будем - расскажу кратко, а подробней будет в блоке, где будут описаны сами стили и протоколы. К гибридным подходам относятся те протоколы, которые совмещают в себе и REST, и RPC, ну или не вписываются ни сюда, ни туда. Так же они позволяют работать с данными и *как с ресурсами* (**REST**), *так с процедурами* (**RPC**) - поэтому и не попадают ни в одну из категорий. Представители - **GraphQL** и **OData**. Чаще всего они используются в *browser-based* фронтенде, где нужно либо тянуть только определённые данные (например через **GraphQL**, где фронтенд может сам выбирать что ему надо, что бы не было *overfetching'a* и фронтенд не нагружал бэкенд + базу данных), или нужны расширенные фильтры для данных (например через **OData**, когда очень тяжело охватывать все

возможности фильтрации в *одном эндпоинте*. **OData** кстати в "*actions*" и "*functions*" умеет, что тоже походит на **RPC**).

Событийные

Этот архитектурный стиль мы уже разбирали в этой статье (блок *Event-Driven*), поэтому тут мы поговорим об этом чуть кратко. *Обмен данных* между сервисами и модулями происходит через *события*. Вместо прямых вызовов один сервис *публикует событие*, а второй сервис (или другие сервис**ы**) *подписываются* на эту *очередь* и *слушают* это *событие*. Так и происходит обмен данными. Такой подход обеспечивает слабую связанность между сервисами и масштабируемость (легко добавить новых "*подписчиков*" не меняя основной код отправителя), *гарантию доставки* (если слушатель упал - событие дождётся пока он восстановится) но при условии правильной конфигурации, ну а так же асинхронность - сервисы не ждут ответа, а они просто публикуют и продолжают работу. Такой подход называется **Pub/Sub моделью** - отправитель не знает получателя, а получатель - не знает отправителя, что как раз и обеспечивают слабую связанность. Применяется в **Highload**, *микросервисной* или **SOA** системах, *legacy-системах*, для *очередей задач* (например обработка изображений или отправка писем), в *realtime-аналитике*. В качестве примера приведу протоколы:

- **AMQP** - Advanced Message Queuing Protocol (**RabbitMQ**)
- **Kafka Protocol** - для стриминга событий (**Apache Kafka**, она кстати может миллионы событий в секунду держать)
- **MQTT**, **NATS** - для IoT, Realtime

Но в этой статье мы их разбирать не будем, ибо их уже в прошлой статье разобрали.

Так, типы протоколов и стилей мы разобрали. А сейчас пойдём уже по конкретным "реализациям". Начнём с самого распространённого - **REST** (да, это не только тип).

HTTP

Но перед тем, как разбирать **REST** - сразу сделаем небольшую ремарку, что нужно понимать разницу между **HTTP API** и **REST API**. Если коротко - то **HTTP API** - это любое **API**, которое строится на базе **HTTP**, и **не важно**, соответствует ли оно принципам (например **REST**) или нет. Грубо говоря - это просто "*API на HTTP*" и не важно, используются ли *ресурсы*, *методы*, *красивые URL* или *глаголы в нейминга*х - главное что общение идёт по **HTTP** протоколу. Например `POST /markdown-create-or-update` - это не **REST**, но вполне себе **API**. Так, теперь продолжим разговор об **REST** и **RESTful**.

REST и RESTful

Как и сказано выше - **Representational State Transfer** (**REST**) самый распространённый архитектурный стиль в web приложениях (*browser-based*, *SPA*,

Mobile Apps), да даже в межсерверном взаимодействии - всякие *external API*, например https://jsonplaceholder.typicode.com/, да и многие другие открытые/закрытые **API**. **REST** - абстрактный архитектурный стиль и набор принципов, который определяет "*как должно быть*". Основные характеристики - всё завязано на ресурсах, stateless модель, поддержка кеширования, и всё, о чём мы писали выше, в блоке с типами про **REST**. Общение может быть как через **JSON**, так и через **XML**, так же поддержка разных *Content-Type* (типов контента) - `application/json`, `application/x-www-form-urlencoded` или `multipart/form-data`. Самое главное - что бы соблюдались принципы "**чистого REST'а по Филдингу**":

- Ресурсная модель: всё - **ресурс с URL**
- Только стандартные **HTTP методы** - `GET`, `POST`, `PUT` и так далее
- **Stateless** - сервер не хранит состояние между запросами
- Поддержка кэшируемости
- **HATEOAS** - клиент узнаёт о действиях через гипермедиа - но на практике почти не реализуется, хотя в идеальном **REST** это важная часть

Так же в **REST** считаются некорректными такие вещи, как:

- Глаголы в URL - `POST /user/create`
- Нарушать **stateless**
- Передавать данные вне тела запроса (например через URL)
- Не делать поддержку кэшируемости

А что же такое **RESTful**? **RESTful по Филдингу** - это **API**, которое полностью соответствует принципам **REST**, то есть "**полное по REST**". Но на практике, в жизни - часто встречаются **API**, которые хоть и заявлены как **RESTful**, но им не являются, и называют их так ради маркетинга, а не ради архитектурной строгости. И скажу больше - чаще всего всего почти всё **API** которое работает через **HTTP** и использует "*красивые*" **URL** называют **REST** или **RESTful**, вне зависимости от того, соответствует ли оно принципам или нет. Настоящий **RESTful** сейчас очень редко можно встретить, но это никого не волнует - главное что бы **API** было удобным и предсказуемым. Например - *Google Maps API* - это вроде бы и **REST**, но часть принципов не используются, не всегда строго ресурсы, а часто "*операции*" как **RPC** подходе, есть очень специфичные методы и параметры, так что *Google Maps API* - это скорее **HTTP API с REST** стилем, называемое **REST-like**. Тут самое главное - знать разницу, не путаться, но и так же понимать, что в современной разработке если слышишь **REST API** - то это может быть и обычное **HTTP API**, и **REST**, и **REST-like**. Использовать этот стиль можно всегда и везде - браузеры его понимают, почти для всех популярных и не очень языков есть **SDK** или библиотеки для **HTTP**, проектировать его легче, да и де-факто - стандарт. Но если требуется что-то, что не входит в стандартную **REST** модель, то можно рассмотреть **RPC подходы**.

Пример структуры

Запрос

```
POST /users HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer <token>
Accept: application/json

{
    "name": "Donh Joe",
    "email": "dohnjoe@mail.com"
}
```

Ответ

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 42,
  "name": "Donh Joe",
  "email": "dohnjoe@mail.com",
  "created_at": "2025-07-12T10:00:00Z"
}
```

Сравнение REST vs RESTful vs REST-like vs HTTP API

Характеристика	**REST**	**RESTful API (на практике)**	**REST-like API**	**HTTP API**
URL	Ресурсы: `/users/42`	Чаще ресурсы, иногда операции	Часто операции	Любые, включая глаголы
HTTP-методы	Строго: `GET` / `POST` / `PUT` / `DELETE`	Обычно да, но бывают отклонения	Любые	Любые
Stateless	Всегда	Часто, но не всегда	Может быть state	Может быть state
HATEOAS	Требуется	Почти нигде	Обычно нет	Нет
Контракт	Не регламентируется, но подразумевается	Любой	Любой	Любой

Характеристика	REST	RESTful API (на практике)	REST-like API	HTTP API
Кэшируемость	Обязательно	Часто игнорируется	Часто нет	Обычно нет
Маркетинг	Не используют	Всегда пишут “RESTful”	Могут писать	Никто не пишет
Типичный пример	API GitHub, учебники	99% современных публичных API	Google Maps API	Внутренние API компаний, кастомные шлюзы

gRPC

gRPC - современный высокопроизводительный **RPC-фреймворк/протокол**, который разработали в *Google*. Его основное предназначение - эффективное взаимодействие между сервисами в распределённых системах и микросервисных архитектурах, когда важны скорость, строгие контракты и поддержка разных языков. Данные, передаются в бинарном виде, и сериализуются на клиенте/сервере по специальным **protobuf**-контрактам (вернее сказать, по сгенерированным структурам на целевых языках и специальному алгоритму). Работает на базе **HTTP/2**, что обеспечивает более быструю передачу и экономию на трафике. Так как в названии есть часть "**RPC**", то логичнее предположить, что этот протокол построен на **RPC принципах**, и это будет чертовски верно. И так как он построен на **RPC** - то соблюдаются все принципы, которые есть в этом самом **RPC**. Ну и куда же без поддержки **TLS/SSL**, легко интегрировать с **OAuth2**, **mTLS**, закрыть токенами через метаданные. Есть даже поддержка стриминга, в том числе и двухстороннего. А контракты строгие, как упоминалось ранее, описаны на специальном языке - **protobuf** (**protocol buffers**), который можно скомпилировать через специальный компилятор **protoc** на поддерживаемые языки, среди которых *Go*, *Python*, *C#*, *Rust* и даже *PHP*. На практике работает это так (сервер):

- Определяем **protobuf** контракт внутри `.proto` файла
- Компилируем сервер из `.proto` файла через **protoc**
- Поднимаем **gRPC** сервер, где определяем процедуры из скомпилированного контракта

А на клиенте это работает вот так:

- Аналогично, из **protobuf** контракта компилируем клиент через **protoc** (на нужном нам языке)
- Определяем клиент, который будет вызывать методы из контракта с параметрами (например `UserService.CreateUser()`)

- Получаем ответ и работаем уже с ним

Так как общий только контракт, можно использовать разные языки для клиента и сервера. Примеры серверов - Go (тут целая микросеврисная архитектура в монорепозитории), PHP, а примеры клиента - Go. А сам **protobuf** выглядит вот так:

```
syntax = "proto3";

package auth;

option go_package = "proto/auth";
option php_namespace = "GRPC\\Auth";
option php_metadata_namespace = "GRPC\\GPBMetadata";

service UserService {
  rpc CreateUser (CreateUserRequest) returns (CreateUserResponse);
}

message CreateUserRequest {
  string name = 1;
  string email = 2;
}

message CreateUserResponse {
  int32 id = 1;
  string name = 2;
  string email = 3;
}
```

Тут всё просто - определяем RPC методы, структуры сообщений и порядок данных, в котором будут данные сериализоваться.

Плюсы **gRPC**:

- Скорость - быстро и с очень низким *overhead*
- Строгая типизация и автогенерация серверов и клиентов, меньше багов на "стыках"
- Стриминг данных в realtime - да-да, всё поддерживается, и даже bidirectional - когда стримить можно в две стороны
- Кросс-языковая поддержка
- Инструменты для мониторинга

Скажу по моему опыту - самый приятный для **gRPC** язык - это *Go*, но на *PHP* тоже можно строить **gRPC** сервер и клиент, но для сервера придётся использовать RoadRunner. А вот минусы этого протокола:

- Это не для людей - сложно дебажить из-за бинарного вида данных, хотя Postman позволяет на лету сериализовать данные в JSON
- В браузерах практически не поддерживается без специального прокси
- Нужно понимать что такое **protobuf**, какие у него ограничения и тому подобное, как в **REST** не получится
- Не все сервисы-посредники (условные прокси типа nginx) на лету поддерживают **HTTP/2** и **gRPC**

Так для чего же выбирать **gRPC**? Всё просто - межсерверные взаимодействия, особенно если система высоконагруженная. Микросервисы, микросервисы в Highload проектах, backend-to-backend. Ну а так же там, где нужен строгий контракт и скорость передачи данных. Для классического *browser-based* фронтенда, да и для любого фронтенда - это будет ту мач, да и практически невозможно. Но для взаимодействия микросервисов - вообще самый раз, красота.

JSON-RPC (XML-RPC)

JSON-RPC (или **XML-RPC**, отличается от **JSON-RPC** только тем, что язык данных - **XML**) - легковесный текстовый протокол удалённого вызова процедур (**RPC**) по сети, в котором сериализация происходит через обычный **JSON**. Он очень прост - клиент отправляет **JSON** с именем *метода* и *параметрами*, а сервер возвращает либо результат, либо ошибку всё в том же **JSON**. Да, некоторые почти-разработчики, уж особенно которые работают с такими *коричневыми* системами как битрикс или вордпресс, могут спросить, так а чем же всё-таки отличаются **JSON-RPC** от **REST**? Ну для таких около-разработчиков и почти-кодеров объясняю - разные подходы. **JSON-RPC** имеет более чёткий контракт (пример будет чуть позже), оперирует не ресурсами (`POST /user`, а `CreateUser(name, email)`), благодаря чему можно можно строить более "точечные" **API**-системы. Зачастую **JSON-RPC** строится на базе **HTTP** (хотя можно и на базе **WebSocket** или **TCP** построить, в реальности чаще всего можно встретить реализацию на **WebSockets**), где есть один эндпоинт, например `POST /rpc`, через который и происходят все вызовы. Ну и более строгий контракт, вот кстати как он выглядит:

Запрос

```
POST /rpc HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "jsonrpc": "2.0",
  "method": "subtract",
  "params": [15, 5],
  "id": 1
}
```

Ответ

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

{
  "jsonrpc": "2.0",
  "result": 10,
  "id": 1
}
```

Вот так всё просто - контракт строгий, результат тоже. Ну и ещё один важный момент - **JSON-RPC** позволяет принимать ещё и *batch-запросы* в контексте одного **HTTP** запроса:

Запрос

```
POST /rpc HTTP/1.1
Host: example.com
Content-Type: application/json

[
    { "jsonrpc": "2.0", "method": "sum", "params": [1,2,3,4,5], "id": "1"
},
    { "jsonrpc": "2.0", "method": "subtract", "params": [15, 10], "id":
"2" }
]
```

Ответ

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

[
    {"jsonrpc": "2.0", "result": 15, "id": 1},
    {"jsonrpc": "2.0", "result": 5, "id": 2}
]
```

Это позволяет сразу отправлять целый пулл каких-либо операций, и экономить на трафике и времени (не забываем, что каждый **HTTP** запрос требует хендшейка, да и всё это время). Ответы кстати тоже более строгие - либо есть "*result*", либо "*error*", ну и так же код ошибки. Кстати, вот и они:

Код	Название	Описание
-32700	Parse error	Ошибка парсинга JSON. Отправлен некорректный JSON.

Код	Название	Описание
-32600	Invalid Request	Неправильный формат запроса (не соответствует спецификации JSON-RPC).
-32601	Method not found	Метод не найден (метода с таким именем нет у сервера).
-32602	Invalid params	Неправильные параметры запроса (не совпадает тип, количество и т.д.).
-32603	Internal error	Внутренняя ошибка сервера JSON-RPC (неожиданная, не относящаяся к бизнес-логике).

И вот как они выглядят в ответе:

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32601,
    "message": "Method not found"
  },
  "id": 1
}
```

Можно кстати определять свои ошибки - в диапазоне между *-32000* и *-32099*. Если обобщить - то это очень хороший стиль для реализации **RPC**, и использовать его можно там, где **REST'a** уже мало, а полноценный **gRPC** - либо долго, либо вообще невозможно (frontend например), очень хороший компромисс. Да и поддержка *batch-запросов* может быть решающим фактором в более нагруженных системах. Используется этот стиль как и в межсерверном взаимодействии, когда контракты описывать долго/бессмысленно, так и в классических *backend+frontend* приложениях, когда **REST'a** уже, как говорилось ранее, мало. Ну или когда нужны *batch запросы*. Кароче, рекомендую хотя бы раз попробовать - гарантирую, вам понравится.

SOAP

SOAP - он же **Simple Object Access Protocol** - это старейший корпоративный протокол обмена данными, который относится к типу **RPC** и который задумывался как "универсальный стандарт" для интеграции между крупными системами. Особенно он был популярным в эпоху "*корпоративных монолитов*" и **ESB**. Основные характеристики:

- **XML-only** - все сообщения и данные сериализуются только в **XML**. Даже самый простой вызов метода превращается в ад из тегов
- Это не только **RPC**, но ещё и протокол документо-ориентированного обмена - передача сложных структур и контрактов

- Строгая спецификация через контракты. **WSDL** (**Web Services Description Langugae**) - специальный **XML-документ**, где прописаны все контракты
- Поддержка сложных сценариев - транзакции, сложная валидация, вложенные вызовы и всё вот это вот
- Можно расширять через заголовки
- Обязательные неймспейсы и строгий контракт, и без этого ничего не заработает
- Не зависит от протокола прикладного уровня - может работать поверх **HTTP**, **FTP**, **SMTP** (даже такое есть, я и сам не знал) - но для каждого свои особенности. В большинстве случаев используется **SOAP over HTTP**

В современных системах **SOAP** чаще всего можно увидеть в старых и "кроваво-энтерпрайзных" проектах, и мои коллеги из моей самой первой работы в галере могут это подтвердить - один крупный белорусский маркетплейс болтов и гаек как раз таки и использовал **SOAP**, что бы данные из 1С в бэкенд мапить. А в моём текущем проекте в одном из стримов солидной бигтех компании тоже используется **SOAP** для передачи данных между основными Core-сервисами, которые написаны очень много лет назад вообще на другом языке, и остальными *backend-for-frontend* сервисами. Сейчас конструировать **SOAP** никто не будет - очень старый протокол, со своими недостатками, а именно:

- Всё завязано на **XML**, из-за чего вырастает избыточность
- Недостаточно просто уметь сериализовать **XML** - нужно ещё и в **WSDL** разобраться, и в namespaces, и в политиках Ws-Security
- Довольно медленный протокол, как раз таки из-за **XML** и сопутствующего оверхэда

Пример **SOAP запроса**:

```
POST /soap HTTP/1.1
Host: example.com
Content-Type: text/xml; charset=utf-8
SOAPAction: "urn:CreateUser"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CreateUser xmlns="http://example.com/users">
      <name>Dohn Joe</name>
      <email>dohnjoe@example.com</email>
    </CreateUser>
  </soap:Body>
</soap:Envelope>
```

Ответ:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CreateUserResponse>
      <id>42</id>
      <name>Dohn Joe</name>
      <email>dohnjoe@example.com</email>
    </CreateUserResponse>
  </soap:Body>
</soap:Envelope>
```

Когда брать **SOAP**? Только в исключительных случаях, когда нужно интегрироваться с очень старыми системами (которые причём сами требуют) и нету других вариантов. Во всех остальных случаях лучше присмотреться к более современным и "оптимизированным" протоколам.

GraphQL

Вот мы и добрались до гибридных типов, и начнём мы с **GraphQL** - протокол для построения **API**, придуманный *корпорацией Цукерберга* для борьбы с ограничениями классического **REST'a**. Самое главное отличие - клиент может сам определять, какие данные ему нужны в нужной структуре, чтобы избежать оверфетчинга и андерфетчинга. И для этого используется специальный язык запроса - **Graph Query Langugae** (отсюда и название - **GraphQL**). Основные характеристики:

- Как и в **JSON-RPC** - единая точка входа, условный `POST /graphql` и все запросы, мутации и подписки идут через него
- Свой язык запросов, описывает какие данные нужно получить, а какие - мутировать
- Строго типизированная схема - вся схема заранее известна и можно автогенерировать типы и автокомплитить поля прямо в IDE
- Есть помимо запросов (**query**) ещё и мутации (**mutation**) - изменения данных, и **subscriptions** - realtime подписки через **WebSocket**
- Можно агрегировать данные из разных источников (БД или внешние **API**, или разные микросервисы), а на выходе получается "единое дерево"
- Batch-запросы ("много в одном") - экономия трафика на мобилках

Почему **GraphQL** можно отнести к гибридному типу? Потому что тут совмещены два подхода - от **REST** мы берём архитектуру, которая строится на ресурсах, только у каждого ресурса своя схема, а от **RPC** - вызовы процедур через **Mutation**, которые не привязаны к ресурсам. Да и мутаторы могут дёргать вообще любые действия над данными, не обязательно работу с ресурсами. Вот пример **GraphQL** запроса:

```
query {
  user(id: 42) {
    name
    email
    orders(limit: 2) {
      id
      price
    }
  }
}
```

Ответ:

```
{
  "data": {
    "user": {
      "name": "Donh Joe",
      "email": "dohnjoe@mail.com",
      "orders": [
        { "id": 1, "price": 1999 },
        { "id": 2, "price": 2999 }
      ]
    }
  }
}
```

Принцип прост - "*получаешь только то, что просишь сам*". Подход идеален для *frontend-heavy*, *SPA* или *мобильных приложений*, в общем где требуются различные куски данных, и классический **REST** не справляется (например требуются разные куски данных под разных пользователей, а тянуть все данные через **REST** - накладно и по расходам на бэкенд, и по расходам на базу, и делать под каждый случай свой эндпоинт - тоже очень не вариант). Если у вас именно такой проект - стоит выбрать именно **GraphQL**. Но и без минусов тоже не обошлось - сложнее проектировать бэкенд сервер, нужно описывать схему, всякие резолверы, валидировать поля доступа на уровне поля или дерева. Ну а так же из-за гибкости могут быть проблемы с кешированием - придётся либо прибегать с специальным прокси в виде **GraphCDN**, или городить свои костыли. Избыточен для простых **CRUD API** - и для "плоских" данных. Но выбирать стоит там, где важна гибкость данных, например сложные дашборды, витрины, или много агрегируемых источников данных. Во остальных случаях лучше рассмотреть другие типы.

OData

Тоже гибридный протокол - **OData** - **Open Data Protocol**, который придумали в *Microsoft* для построения удобных и гибких **API** поверх уже существующих данных - таблиц, моделей или сущностей. Добавляет возможность гибко фильтровать, сортировать, join-ить, пагинировать и так далее, и всё это прямо в стандартном запросе в URL. Грубо говоря - это **REST с query-языком**. Хотя **OData** всё-таки мощнее обычного **REST**. Вот его основные характеристики:

- Как **REST**, только свой язык запросов (**query langugae**). Вся работа идёт через стандартные **HTTP** методы, а параметры запроса позволяют делать любые манипуляции с данными как в **SQL**
- В этом протоколе чётко описана структура данных - **EDMX** (специальная **XML**-схема **Entity Data Model**, на ней часто описывают модели в **.NET**-системах) или **JSON Schema**
- Благодаря описанию схемы можно автогенерировать как доку, так и клиентов на разных языках программирования
- Работает только на **HTTP**, но в качестве сериализации может использовать не только **JSON**, а ещё и **AtomPub** - на базе **XML**

А почему же он тогда тоже гибридный? Всё просто - потому что сочетает в себе и **REST**, и **RPC**. Что, вас такой ответ не устраивает? Ладно, раскрою тему - от **REST'a** он берёт работу с ресурсами и стандартные **HTTP** методы, а под стиль **RPC** - мощный **query-language** и возможность вызывать "*процедуры*". Кароче, сочетает он всё это, как и **GraphQL**. С этим разобрались, теперь **пример**:

```
GET /Products?$expand=Category&$select=ID,Name,Price,Category HTTP/1.1
Host: example.com
```

Ответ (чаще всего в JSON):

```
{
  "@odata.context":
"https://api.example.com/odata/$metadata#Products(ID,Name,Price,Category()
)",
  "value": [
    {
      "ID": 1,
      "Name": "Apple",
      "Price": 1.99,
      "Category": {
        "ID": 100,
        "Name": "Fruits"
      }
    },
    {
      "ID": 2,
```

```
      "Name": "Milk",
      "Price": 2.50,
      "Category": {
        "ID": 200,
        "Name": "Dairy"
      }
    }
  ],
  "@odata.count": 2,
  "@odata.nextLink": "https://api.example.com/odata/Products?$skip=2"
}
```

Как видно - **OData** очень гибкий, как и **GraphQL** позволяет *получить только те данные, которые клиент сам и запросил*, и это без обработки множества кастомных эндпоинтов. Язык запросов очень мощный, фильтрации, сортировки, группировки - всё есть из коробки. Легко интегрироваться с системами от *Microsoft* - они своё детище любят (ну или раньше по крайней мере любили). Есть унификация - все эндпоинты строятся по одной схеме, и открытый стандарт - библиотеки есть под разные языки. Но как же без минусов? А они тоже есть:

- Сложность реализации - тут я думаю всё понятно
- Нужно уметь фильтровать разрешённые доступы, иначе можно случайно слить всю базу данных или пропустить очень дорогой join, который весь бэкенд и повесит
- Специфичность для "табличных" данных - он может не подойти для сложной бизнес-логики или нелинейных моделей

А когда же использовать? Если нужно интегрироваться во всякие "нестандартные" системы, которые и заточены под **OData**, или нужно большие и нелинейные витрины данных (**CRM'ки** всякие с огромными дашбордами, и прочее), или нужно построить **API** с мощной фильтрацией без особой головной боли. А во всех остальных случаях лучше присмотреться к более "*стандартным*" стилям или протоколам. Не стоит изобретать **OData** там, где нужен обычный **REST**.

WebSocket

Самый "необычный" среди всех протокол - он не завязан ни на **HTTP**, ни на **HTTP/2**, и при этом не относится к **брокеру очередей**. Встречайте - **WebSocket**. Построенный на базе **TCP**, имеет свой протокол `ws`, предназначен для *realtime стриминга* (в том числе и *bidirectional*), поддерживается как браузерами, так и мобильными клиентами (за ведро не знаю, но iOS и Flutter/Dart поддерживают). Кароче - очень полезный, мощный протокол, на котором можно (нужно) реализовывать *realtime системы*. Так как он не имеет привычного для **HTTP** (который к слову тоже на базе **TCP** построен) хендшейка на каждый запрос - можно единоразово открыть постоянное соединение и слать/слушать данные независимо друг от друга. Самые основные особенности:

- Постоянный коннект - соединение живёт от инициации (открытия) и до явного закрытия или обрыва
- Независимый двусторонний обмен данными - как клиент, так и сервер могут независимо отправлять данные друг другу когда угодно
- Overhead минимальный - меньше заголовков, высокая скорость из-за отсутствия постоянных хендшейков
- Нету ограничения на формат данных - можно отправлять как **JSON**, так и строки, бинарные данные (через MsgPack например)

А работает **WS** по простому принципу:

- Клиент инициирует **upgrade запрос** по **HTTP** (**HTTPS**), а сервер в свою очередь апрвувает переход на **WebSocket**-протокол
- После установления канала клиент и сервер могут посылать данные независимо друг от друга. Чаще встречаются ещё реализация *ping-pong* сообщений раз в n-секунд, что бы проверять что "клиент" живой и его не нужно выкидывать из пулла подключений на сервере - клиент просит "ping" - сервер понимает что клиент живой и отвечает "pong", а если за указанное n секунд клиент ping не прислал - удаляем его подключение (делаем ему дисконнект)

Пример работы **WS**:

Handshake (обычно браузер делаем сам, сервер сам его обрабатывает)

```
GET /notification HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: <random>
Sec-WebSocket-Version: 13
```

Теперь можно слать любые пакеты (*бинарка/текст/текст в JSON*)

Клиент:

```
{
  "message": "ping"
}
```

Сервер

```
{
  "message": "pong"
}
```

```
{
  "type": "error",
  "message": "User registered"
}
```

И, опять же, можно отправлять пакеты, которые по-разному сериализованны (даже в одном коннекте, если сервер это поддерживает)

Клиент (MsgPack + base64)

```
gadtZXNzYWdlpHBpbmc=
```

Сервер (MsgPack + base64)

```
gadtZXNzYWdlpHBvbmc=
```

В этом примере отправляются те же сообщения с *ping-pong*, что и в примере с **JSON**, только это всё через **MsgPack** закодировано. Также, ещё одной главной особенностью является то, что на базе **WS** протокола можно реализовывать какие угодно контракты, главное - что бы была согласованность между клиентом и сервером. Так же можно на базе **WS** реализовать **JSON-RPC**, что будет даже более правильным, можно сэкономить на транспорте, или даже **GraphQL** построить. Ну и можно свой *Event-Driven* подход построить не через классические для этого подхода **AMQP** или **Kafka**, а через **WS**. Тут, как говорится, флаг вам в руки и барабан на шею. А для чего используется этот протокол? Для всего, где нужно *realtime* связь - нотификации, они же уведомления в реальном времени (да, битриксоиды-кубоиды, камень в ваш огород - не нужно делать пуллинг сервера по так называемому в вашем кругу "ajax" на предмет новых уведомлений, достаточно использовать подходящие для этого инструменты), онлайн-чаты (это уже камень в сторону ВК, в веб версии насколько я помню до сих пор лонг-пуллинг используется, хотя надеюсь я ошибаюсь. Хотя нет, не надеюсь), биржы всякие, короче - везде где нужен *realtime*. Например дискорд всю свою коммуникацию построил на веб-сокетах. Также, стоит рассмотреть **WS** если вам нужно передавать часто/много данных, ибо можно существенно сэкономить на транспорте, нежели использовать классические **REST** или **HTTP API**. Но а теперь по минусам - как и в любой вещи в мире, этот протокол тоже имеет минусы:

- Сложнее масштабировать - ибо для каждого соединения нужно хранить **state connection**, и на highload системах для горизонтального масштабирования придётся что-то придумывать через брокер сообщений или общую шину данных (как пример - мой CV генератор в микросервисной архитектуре, где WS слушает RabbitMQ внутри API-Gateway, а Generator Service - из RabbitMQ получает данные для генерации, генерирует PDF и через тот же RabbitMQ отправляет событие на

gateway для уведомление юзера. Да, не совсем про это конечно, но логика работы схожая)

- Нужно реализовывать или контролировать авторизацию, timeouts, защиту всякую от флуда и прочего. Laravel Reverb например для проверки авторизации использует отдельный `POST /broadcast/auth` - вроде бы так, точно не помню, и от него уже даёт апрув или реджект на создание **WS** подключения (да, *Laravel Reverb* построен на базе **WebSocket**)
- К минусам можно отнести ещё отсутствие единого стандарта **API** - каждый "*делает то, о чём писал великий Бродский*" как он хочет
- На некоторых языках или фреймворках построить **WebSocket** сервер - довольно не тривиальная задача, и **PHP** входит в этот список. На чистом **PHP** - легче закосплеить Кобейна (начиная с его 13-ти лет, и потом как только не получится - то уже закосплеить его 5 августа 1994 года), чем поднять его на чистом **PHP**. Но отчаиваться не стоит - можно использовать Swoole extencion, но с этим тоже будет довольно сложно, а можно взять Centrifugo, но тогда фокус боли сменится с реализации **WS** на конфигурацию центрифуги. Самый лучший вариант - использовать RoadRunner + Centrifugo - избавляет от части головной боли. Да и вообще использовать **RoadRunner** в своём проекте - скорее good, чем bad, хотя нужно смотреть по потребностям

И так, если коротко - то **WS** вам нужны только в случае, если вам надо сделать *realtime систему*, или у вас очень много или часто данными обмениваются при больших нагрузках, и можно на этом сэкономить. Во всех остальных случаях - возьмите **REST** или что-то из списка выше, что будет соответствовать вашим задачам. Но и городить велосипед из классического **HTTP** не стоит там, где явно нужны **WebSocket'ы**. Может выйти очень дорого.

Вот и всё - мы разобрали все основные стили и протоколы. Мы выявили плюсы и минусы каждого. Но наблюдательный читатель может заметить, что мы обошли стороной **событийные** типы - и верно заметит. Всё дело в том, что мы их разбирали в прошлой статье (в части номер один) и немного затронули тут, и больше мне сказать о них нечего - ну кроме того, что их стоит использовать в межсерверном общении по принципу "*паблиш эвент*". Ну а более прошаренные или опытные читатели справедливо заметят, что не упомянута такая тема, как реализация своих протоколов поверх **TCP** или **UDP**. Для таких читателей поясняю - это довольно нишевая тема, и люди, которым это необходимо не нуждаются в моих советах и объяснениях, эти люди сами мне пояснят за все протоколы, без подсказок и скорее всего с явным презрением. А если резюмировать то, что в этой статье мы разбирали - при проектировании вашего приложения отталкивайтесь от логики "**HTTP/REST API first**", и дальше методом исключения выбирайте нужный вам стиль/протокол в зависимости от того, что вы проектируете, для чего и что должно получится. Иногда лучше сразу заложить **JSON-RPC** или **GraphQL**, чем потом переписывать всё с **REST'а**, или заранее описать **protobuf'ы** и построить общение между микросервисами через **gRPC**. Ну и для *realtime* - **WebSockets**, но для *backend+frontend*. А для межсерверного взаимодействия

по принципу "**событийности**" - **RabbitMQ** или **Kafka** (если понимаете, зачем она вам. Иногда нужен "*кролик*", просто "*кролик*"). И ещё проверяйте документацию и зрелость инструментов - а то можно взять либо новый и "непроверенный" инструмент, который поможет обрести геморрой когда в проде выстрелит непонятным багом (и оценивайте свои силы - вряд ли вы пойдёте форкать либу и самим фиксить этот баг в форке. Так себе только бигтехи позволить могут), или слишком старый инструмент, под которого сейчас разработчика и днём с огнём не сыщешь. А если и сыщешь - то будет стоить больше, чем ваша двушка в Алтуфьево. И никогда не ищите *серебряную пулю* - **каждый инструмент имеет свои плюсы и минусы**, **каждый инструмент решает свою задачу**. Таблетки "*от всего*" **не существует**, и подходить к выбору инструмента нужно осознанно, с анализом, а если анализ показывает, что "*всё, что не REST - не нужен*", значит действительно, ***всё, что не REST - не нужен***. И ещё всё-таки стоит сверяться с best practices (для кубоноидов-битриксоидов - "лучшие практики") и реальными кейсами, технологии меняются быстро, а вам ещё эту систему поддерживать. А за сим я с вами прощаюсь, надеюсь мой цикл статей был вам полезен. Сравнительная таблица будет ниже, по ней легче будет ориентироваться.

Тип	**Протокол/ Стиль**	**Краткое описание**	**Примеры/ Транспорт**
REST	RESTful API	Ресурсно-ориентированный API. Операции с сущностями через стандартные HTTP-методы.	HTTP, JSON, XML
RPC	gRPC	Процедурный вызов удалённых методов. Использует protobuf, работает поверх HTTP/2, строгие схемы.	HTTP/2, ProtoBuf
RPC	JSON-RPC	Лёгкий RPC через JSON. Все вызовы — вызов метода с параметрами.	HTTP, WebSocket, JSON
RPC	XML-RPC	То же, что JSON-RPC, но через XML.	HTTP, XML
RPC	SOAP	Формальный протокол, строго типизированный, всегда XML, поддерживает сложные контракты.	HTTP, XML
Гибридный	GraphQL	Query-based API. Клиент формирует запросы на нужные данные.	HTTP, WebSocket, JSON
Гибридный	OData	REST + Query language. Позволяет гибко фильтровать, сортировать, выбирать ресурсы через параметры	HTTP, JSON, AtomPub/ JSON
Событийный	AMQP (RabbitMQ)	Message Queue протокол. Передача сообщений между	TCP, AMQP

Тип	Протокол/ Стиль	Краткое описание	Примеры/ Транспорт
		сервисами, Pub/Sub, очереди.	
Событийный	Kafka	Event Streaming. Масштабируемая шина событий для микросервисов и аналитики.	TCP, Kafka protocol
Событийный	NATS, MQTT	Лёгкие Pub/Sub и IoT-протоколы для событий, сообщений, сигнализации.	TCP, WebSocket, MQTT, NATS
Гибридный/ Событийный	WebSocket (event)	Двусторонний realtime-канал. Передача событий, сигналов, пуш-уведомлений.	TCP, WebSocket, JSON