**JSON-RPC и XML-RPC это одно и тоже. Просто разный формат данных. Ниже мы поговорим о JSON-RPC, так как его формат наиболее удобно читать**

Это простой stateless-**протокол** для создания API в стиле RPC (Remote Procedure Call).  
Выглядит это обычно следующим образом

У вас на сервере есть один единственный endpoint, который принимает запросы с телом вида:



И отдает ответы вида:



Если возникает ошибка — ответ об ошибке:



И это всё!

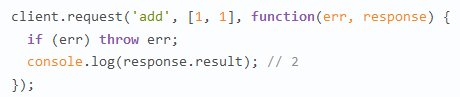
Бонусом поддерживаются batch-операции:



В поле id клиент API может отправлять что угодно, дабы после получения ответов от сервера сопоставить их с запросами.  
  
Также клиент может отправлять «нотификации» — запросы без поля «id», которые не требуют ответа от сервера:



Библиотеки для клиента и сервера есть, наверное, под все популярные языки.  
Если нет — не беда. Протокол настолько простой, что написать свою реализацию займет пару часов.  
  
Работа с RPC-клиентом, который мне первым попался на npmjs.com, выглядит так:



**Плюсы**

**Согласованность с бизнес-логикой проекта**  
  
Во-первых, можно не прятать сложные операции за скудным набором HTTP-глаголов и избыточными URI.  
  
Есть предметные области, где операций в API должно быть больше чем сущностей.  
Навскидку — проекты с непростыми бизнес-процессами, gamedev, мессенджеры и подобные realtime-штуки.  
  
Да даже взять контентный проект вроде Хабра…  
  
Нажатие кнопки "↑" под постом — это не изменение ресурса, а вызов целой цепочки событий, вплоть до выдачи автору поста значков или инвайтов.  
  
Так стоит ли маскировать post.like(id) за PUT /posts/{id}/likes?  
  
Здесь также стоит упомянуть [CQRS](https://martinfowler.com/bliki/CQRS.html), с которым RPC-шный API будет смотреться лучше.  
  
Во-вторых, кодов ответа в HTTP всегда меньше, чем типов ошибок бизнес-логики, которые вы бы хотели возвращать на клиент.  
  
Кто-то всегда возвращает 200-ку, кто-то ломает голову, пытаясь сопоставить ошибки с HTTP-кодами.  
  
В JSON-RPC весь диапазон integer — ваш.

**JSON-RPC — стандарт, а не набор рекомендаций**



POST /server/{id}/status или PATCH /server/{id}?  
Это больше не имеет значения. Остается POST /api.

Нет никаких best practices с форумов, есть стандарт.  
Нет разногласий в команде, есть стандарт.

Конечно же, качественно реализованный REST API можно полностью задокументировать. Однако…  
  
Знаете, что и где нужно передать в запросе к Github API, чтобы получить объект reactions вместе с issue?



Хорошо это или плохо? Решайте сами, гуглите сами. Стандарта нет.

**Независимость от HTTP**

В теории, принципы REST можно применять не только для API поверх HTTP.  
На практике все по-другому.

JSON-RPC over HTTP безболезненно переносится на JSON-RPC over Websocket. Да хоть TCP.  
Тело JSON-RPC запроса можно прямо в сыром виде бросить в очередь, чтобы обработать позже

Больше нет проблем от размазывания бизнес-логики по транспортному уровню (HTTP).



**Производительность**

JSON-RPC пригодится, если у вас есть:  
— Batch-запросы  
— Нотификации, которые можно обрабатывать асинхронно  
— Вебсокеты  
  
Не то, чтобы это все нельзя было сделать без JSON-RPC. Но с ним — чуть легче

**Подводные камни**

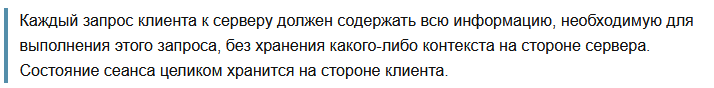
**HTTP-кеширование**  
  
Если вы собираетесь кешировать ответы вашего API на уровне HTTP — RPC может не подойти.  
Обычно это бывает, если у вас публичное, преимущественно read-only API.  
Что-то вроде получения прогноза погоды или курса валют.  
  
Если ваше API более «динамичное» и предназначено для «внутреннего» использования — все ок.

**access.log**  
  
Все запросы к JSON-RPC API в логах веб-сервера выглядят одинаково.  
Решается логированием на уровне приложения.

**Документирование**  
  
Для JSON-RPC нет инструмента уровня [swagger.io](https://swagger.io/).  
Подойдет [apidocjs.com](http://apidocjs.com/), но он гораздо скромнее.  
Впрочем, документировать такой простой API можно хоть в markdown-файле.

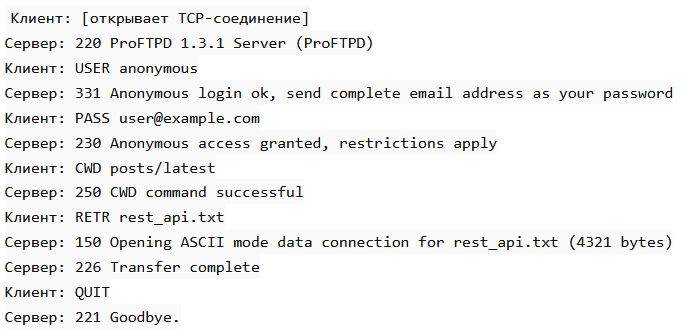
**Stateless**  
  
«REST»  — об архитектуре, а не глаголах  HTTP — возразите вы. И будете правы.

В оригинальной диссертации Роя Филдинга не указано, какие именно глаголы, заголовки и коды HTTP нужно использовать.  
  
Зато в ней есть волшебное слово, которое пригодится даже при проектировании RPC API. «Stateless».



Делая RPC API поверх веб-сокетов, может возникнуть соблазн заставить сервер приложения хранить чуть больше данных о сессии клиента, чем нужно.

Насколько stateless должен быть API, чтобы не причинять проблем? Для контраста вспомним по-настоящему statefull протокол  —  FTP.



Состояние сеанса хранится на сервере. FTP-сервер помнит, что клиент уже прошел аутентификацию в начале сеанса, и помнит, в каком каталоге сейчас «находится» этот клиент.  
  
Такой API сложно разрабатывать, дебажить и масштабировать. Не делайте так.

**В итоге**

Возьмите JSON-RPC 2.0, если решитесь сделать RPC API поверх HTTP или веб-сокетов.  
Можете, конечно, придумать свой велосипед, но зачем?  
  
Возьмите GraphQL, если он правда вам нужен.  
  
Возьмите gRPC или что-то подобное для коммуникации между (микро)сервисами, если ваш ЯП это поддерживает.  
  
Возьмите REST, если нужен именно он. Теперь вы, по крайней мере, выберете его осознанно.