Принципы REST

1. Client-Server
2. Stateless
3. Cacheable
4. Uniform interface
5. Layered system
6. Code on demand

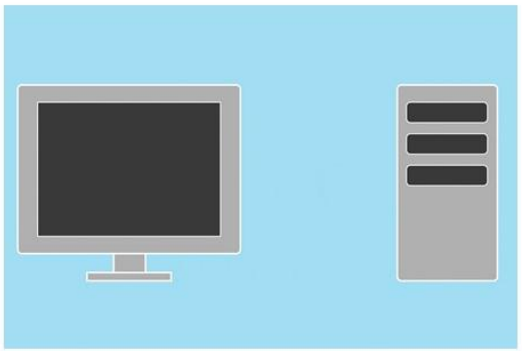
Существует 6 принципов написания RESTful-интерфейсов.

1.Client-Server

2. Stateless
3. Cacheable
4. Uniform interface
5. Layered system
6. Code on demand

Приведение архитектуры к модели клиент-сервер

Необходимо отделять потребности клиентского интерфейса от потребностей сервера, хранящего данные. Само разграничение на "клиент" и "сервер" позволяет им развиваться независимо друг от друга.



Client-Server.

В основе данного ограничения лежит разграничение потребностей. Необходимо отделять потребности клиентского интерфейса от потребностей сервера, хранящего данные. Данное ограничение повышает переносимость клиентского кода на другие платформы, а упрощение серверной части улучшает масштабируемость системы. Само разграничение на “клиент” и “сервер” позволяет им развиваться независимо друг от друга.
Отделяя пользовательский интерфейс от хранилища данных, мы улучшаем переносимость пользовательского интерфейса на другие платформы и улучшаем масштабируемость серверных компонент засчёт их упрощения.

Приложение не должно беспокоиться о том как сервер сохраняет данные, особенности и тип базы данных, техническая часть и т.д Точка взаимодействия между сервером и приложением – схема базы данных, тоесть что и где храниться. Эта схема часть Апи документации.

Два самых важных пуктов взаимодействия между клиетом и сервером –
1.Формат данных, который нужен клиенту
Сервер должен поддерживать формат данных, который нужен клиенту, иначе взаиможействие будет безполезным.
Приложение использует данные, а апи продукт. Для того чтобы этот продует был успешным он должен удовлетворять потребности клиента.
Например если клиенту нужны картинки в формате jpeg это значит что бекенд должен их доставлять. Если на сервере храняться картинки в формате RAW то нужно предусмотреть то, что их конвертируют клиенту в формате JPEG
Клиент должен заявлять серверу о желаемом формате. Например – /api/user/id/profilepicture.jpg

2.Предоставлять только нужную информацию. Если клиенту нужна одна фотография, сервер не должен высылать весь альбом. Мораль сей басни такова - чтобы не перегружать трафик – минимизировать трафик и максимизировать пропускную способность) — объем информации, который способен пропустить канал за определенный временной промежуток

1. Client-Server
2. **Stateless**
3. Cacheable
4. Uniform interface
5. Layered system
6. Code on demand

Отсутствие состояния

Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Таким образом и сервер, и клиент могут "понимать" любое принятое сообщение, не опираясь при этом на предыдущие сообщения.

Stateless (без состояния). Архитектура REST требует соблюдения следующего условия. В период между запросами серверу не нужно хранить информацию о состоянии клиента и наоборот. Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Таким образом и сервер, и клиент могут "понимать" любое принятое сообщение, не опираясь при этом на предыдущие сообщения.
Каждый запрос от клиента к серверу должен содержать в себе всю необходимую информацию и не может полагаться на какое-либо состояние, хранящееся на стороне сервера. Таким образом, информация о текущей сессии должна целиком храниться у клиента.

Если это было бы в реальной жизни то оно звучало бы так =

Где живет Вадим? Сколько ему лет?

В первом вопросе предполагается что мы должны ответить где живет Вадим. А второй вопрос понятно что идет речь о Вадиме и мы спрашиваем сколько ему лет.

В REST API, такой же диалог будет звучать так:

Где живет Вадим?

Сколько Вадиму лет?

1. Client-Server
2. Stateless
3. **Cacheable**
4. Uniform interface
5. Layered system
6. Code on demand

Кэширование

Клиенты могут выполнять кэширование ответов сервера. У тех, в свою очередь, должно быть явное или неявное обозначение как кэшируемых или некэшируемых, чтобы клиенты в ответ на последующие запросы не получали устаревшие или неверные данные.

Cacheable (кэшируемость). Это ограничение требует, чтобы для данных в ответе на запрос явно было указано -- можно их кэшировать или нет. Если ответ поддерживает кэширование, то клиент имеет право повторно использовать данные в последующих эквивалентных запросов без обращения на сервер.

Клиенты могут выполнять кэширование ответов сервера. У тех, в свою очередь, должно быть явное или неявное обозначение как кэшируемых или некэшируемых, чтобы клиенты в ответ на последующие запросы не получали устаревшие или неверные данные. Правильное использование кэширования помогает полностью или частично устранить некоторые клиент-серверные взаимодействия, ещё больше повышая производительность и расширяемость системы.

1. Client-Server
2. Stateless
3. Cacheable
4. **Uniform interface**
5. Layered system
6. Code on demand

Единообразие интерфейса

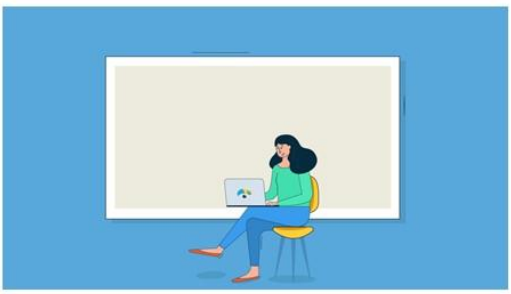
Клиент должен всегда понимать, в каком формате и на какие адреса ему нужно слать запрос, а сервер, в свою очередь, также должен понимать, в каком формате ему следует отвечать на запросы клиента. Этот единый формат клиент-серверного взаимодействия, который описывает, что, куда, в каком виде и как отсылать и является унифицированным интерфейсом

Uniform interface (единообразие интерфейса). Если применить к систем инженерный принцип общности/единообразия, то архитектура всего приложения станет проще, а взаимодействие станет прозрачнее и понятнее. Для выполнения этого принципа необходимо придерживаться нескольких архитектурных ограничений.
Клиент должен всегда понимать, в каком формате и на какие адреса ему нужно слать запрос, а сервер, в свою очередь, также должен понимать, в каком формате ему следует отвечать на запросы клиента. Этот единый формат клиент-серверного взаимодействия, который описывает, что, куда, в каком виде и как отсылать и является унифицированным интерфейсом

1. Client-Server
2. Stateless
3. Cacheable
4. Uniform interface
5. **Layered system**
6. Code on demand

Слои

Под слоями подразумевается иерархическая структура сетей. Иногда клиент может общаться напрямую с сервером, а иногда — просто с промежуточным узлом. Применение промежуточных серверов способно повысить масштабируемость за счёт балансировки нагрузки и распределённого кэширования.

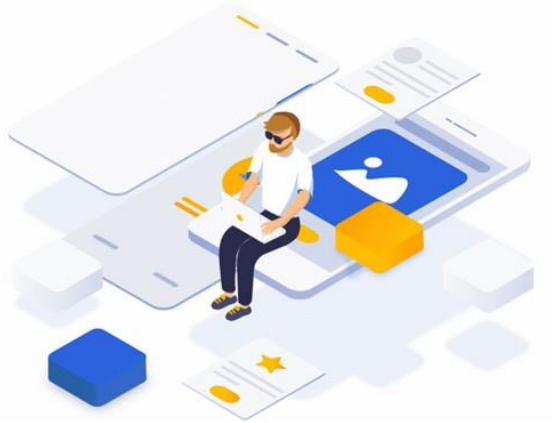


Layered system (многоуровневая система). Многоуровневость достигается засчёт ограничения поведения компонентов таким образом, что компоненты "не видят" другие компоненты, кроме расположенных на ближайших уровнях, с которыми они взаимодействуют.
Под слоями подразумевается иерархическая структура сетей. Иногда клиент может общаться напрямую с сервером, а иногда — просто с промежуточным узлом. Применение промежуточных серверов способно повысить масштабируемость за счёт балансировки нагрузки и распределённого кэширования.
Приведем пример. Представим себе некоторое мобильное приложение, которое пользуется популярностью во всем мире. Его неотъемлемая часть — загрузка картинок. Так как пользователей — миллионы человек, один сервер не смог бы выдержать такой большой нагрузки. Разграничение системы на слои решит эту проблему. Клиент запросит картинку у промежуточного узла, промежуточный узел запросит картинку у сервера, который наименее загружен в данный момент, и вернет картинку клиенту. Если здесь на каждом уровне иерархии правильно применить кэширование, то можно добиться хорошей масштабируемости системы.

1. Client-Server
2. Stateless
3. Cacheable
4. Uniform interface
5. Layered system
6. **Code on demand**

Код по требованию (необязательное ограничение)

Данное ограничение подразумевает, что клиент может расширять свою функциональность, за счет загрузки кода с сервера в виде апплетов или сценариев, скриптов



Code on demand (код по мере необходимости, необятельно). REST позволяет наращивать функциональность клиентского приложения по мере необходимости при помощи скачивания и исполнения кода в виде апплетов или скриптов. Это упрощает клиентские приложения, уменьшая количество заранее написанных возможностей.

POST -- create
GET -- read
PUT -- update/replace
DELETE -- delete
PATCH -- partial update/modify
Данное ограничение подразумевает, что клиент может расширять свою функциональность, за счет загрузки кода с сервера в виде апплетов или сценариев.

Только что употребленный термин **RESTful** (веб-)сервис всего лишь означает сервис, реализованный с использованием принципов REST. **Так что же нам дает следование этим самым принципам REST?** Для начала я бы назвал простоту основным преимуществом архитектуры REST. Простоту идеи, простоту разработки и добавления функциональности к RESTful приложениям. Идея настолько проста и универсальна, что ее даже сложно сначала уловить. Мы не добавляем никакого нового слоя в наш и без того многослойный программерский пирог, а просто используем уже давно признанные стандарты.

Источники

1. https://coderlessons.com/tutorials/veb-razrabotka/arkhitektura-veb-servisov/8-api-protiv-veb-sluzhby
2. https://overcoder.net/q/442424/%D0%B2-%D1%87%D0%B5%D0%BC-%D1%80%D0%B0%D0%B7%D0%BD%D0%B8%D1%86%D0%B0-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-api-%D0%B8-%D0%B2%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81%D0%B0%D0%BC%D0%B8
3. https://www.intervolga.ru/blog/projects/relsy-veb-integratsii-rest-i-soap/
4. https://coderlessons.com/tutorials/veb-razrabotka/izuchite-veb-servisy/veb-servisy-kratkoe-rukovodstvo
5. https://habr.com/ru/post/46374/
6. https://medium.com/future-vision/the-principles-of-rest-6b00deac91b3
7. https://javarush.ru/groups/posts/2486-obzor-rest-chastjh-1-chto-takoe-rest
8. https://javarush.ru/groups/posts/2487-obzor-rest-chastjh-2-kommunikacija-mezhdu-klientom-i-serverom-
9. https://habr.com/ru/post/131343/
10. https://coderlessons.com/tutorials/veb-razrabotka/arkhitektura-veb-servisov/4-veb-servisy-restful
11. https://habr.com/ru/post/483202/