Руководство по REST API

Что такое REST?

Архитектура REST описывается шестью ограничениями. Эти ограничения, применительно к архитектуре, первоначально были представлены Роем Филдингом (Roy Fielding) в его докторской диссертации и определяют основы RESTful стиля.

Шесть ограничений:

Единый интерфейс

Единый интерфейс определяет интерфейс между клиентами и серверами. Это упрощает и отделяет архитектуру, которая позволяет каждой части развиваться самостоятельно. Четыре принципа единого интерфейса:

Основан на ресурсах
Отдельные ресурсы определяются в запросе, для чего используется URI, как идентификаторы ресурсов. Сами ресурсы концептуально отделены от представлений, которые возвращаются клиенту. Например, сервер не отправляет свою базу данных, а, скорее, некоторые HTML, XML или JSON, которые представляет некоторые записи в базе данных, например, на финском языке и в UTF-8, в зависимости от деталей запроса и реализации сервера.

Манипуляции над ресурсами через представления
Когда пользователь имеет представление о ресурсе, в том числе о связанных метаданных, он имеет достаточно информации для изменения или удаления ресурса на сервере, если у него есть на это разрешение

Самодокументируемые сообщения
Каждое сообщение содержит достаточно информации для описания того, как его выполнить. Например, вызываемый парсер может описываться с помощью Internet media type (так же известным как MIME) Ответы также явно указывают на их способность кешировать.

Hypermedia as the Engine of Application State (HATEOAS)
Клиенты предоставляют статус через содержимое body, параметры строки запроса, заголовки запросов и запрашиваемый URI (имя ресурса). Это называется гипермедиа (или гиперссылки с гипертекстом)

Наряду с приведенным выше описанием, HATEOAS также означает, что, в случае необходимости ссылки содержатся в теле ответа (или заголовках) для поддержки URI извлечения самого объекта или запрошенных объектов. Позднее, мы затронем эту тему глубже.

Единый интерфейс так же означает, что любой REST сервис должен обеспечивать его фундаментальный дизайн.

Отсутствие состояний (Stateless)

Так как REST это акроним для REpresentational State Transfer, отсутствие состояний является важной чертой. Таким образом, это значит, что необходимое состояние для обработки запроса содержится в самом запросе, либо в рамках URI, параметрах строки запроса, тела или заголовках. URI уникально идентифицирует ресурс, и тело содержит состояние (или изменение состояния) этого ресурса. Затем, после того, как сервер завершит обработку, состояние или его часть(и) отдаётся обратно клиенту через заголовки, статус и тело ответа.

Большинство из нас, кто был в этой отрасли, привыкли к программированию в контейнере, который даёт нам понятие «Сессия, которая поддерживает состояние нескольких HTTP запросов. В REST, клиент должен включать всю информацию для сервера для выполнения запроса, по необходимости повторно отправляя состояние, если это состояние должно охватывать несколько запросов. Отсутствие состояний обеспечивает большую масштабируемость, так как сервер не должен поддерживать или общаться через состояние сеанса. Кроме того, балансировщику нагрузки не придётся беспокоиться о связанности сессии и системы.

Так в чём различие между состоянием и ресурсом? Состояние или состояние приложения, это то, что сервер заботится выполнить запрос для получения данных необходимых для текущей сессии или запроса. Ресурсное состояние, или ресурс, это данные, которые определяют представление ресурса, например, данные хранящиеся в базе данных. Рассмотрим состояние приложения как данные, которые могут варьироваться в зависимости от клиента и запроса. С другой стороны, состояние ресурсов постоянно по каждому клиенту, который запрашивает его.

Каждый встречал проблему с кнопкой «Назад» в своём веб приложении, когда оно ведет себя по-разному в одной точке, потому что ожидались действия в определенном порядке? Такое происходит, когда нарушен принцип отсутствия состояний. Есть случаи, когда не соблюдается принцип отсутствия состояний, например, three-legged OAuth, ограничение скорости вызова API и т.д. Однако, приложите максимум усилий, чтобы состояние приложения не занимало несколько запросов к вашему сервису.

Кеширование ответа (Cacheable)

Как и в World Wide Web, клиент может кэшировать ответы. Таким образом, ответы явно или неявно определяют себя как кешируемые или нет, для предотвращения повторного использования клиентами устаревших или некорректных данных в ответ на дальнейшие запросы. Хорошо спроектированное кэширование частично или полностью устраняет некоторые клиент-серверные взаимодействия, способствуя дальнейшей масштабируемости и производительности.

Клиент-сервер (Client-Server)

Единый интерфейс отделяет клиентов от серверов. Разделение интерфейсов означает, что,

например, клиенты не связаны с хранением данных, которое остаётся внутри каждого сервера, так что мобильность кода клиента улучшается. Серверы не связаны с интерфейсом пользователя или состоянием, так что серверы могут быть проще и масштабируемы. Серверы и клиенты могут быть заменяемы и разрабатываться независимо, пока интерфейс не изменяется.

Многоуровневая система (Layered System)

Обычно клиенты не могу сказать — они подключены напрямую к серверу или общаются через посредника. Промежуточный сервер может улучшить масштабируемость системы, обеспечивая балансировку нагрузки и предоставляя общий кэш. Слои также могут отвечать за политику безопасности.

"Код по требованию" (Code on Demand - опционально)

Серверы могут временно расширять или настраивать функциональность клиента, передавая ему логику, которую он может исполнять. Например, это могут быть скомпилированные Java-апплеты или клиентские скрипты на Javascript

Соблюдая эти ограничения, и, таким образом, придерживаясь RESTful архитектуры, мы позволяем распределенной системе любого типа иметь такие свойства как: производительность, расширяемость, простота, обновляемость, понятность, портативность и надёжность.

Замечание Единственным необязательным ограничением для RESTful архитектуры — это «код по требованию». Если сервис не проходит по любым другим условиям, то его совершенно точно нельзя назвать RESTful.

Советы по REST API

Будь то RESTful или нет (в соответствии с шестью ограничениями, описанными ранее), вот несколько рекомендованных REST концепций, которые помогут построить более хорошие и удобные сервисы:

Используйте HTTP-глаголы, чтобы ваши запросы имели понятное значение

Пользователи API должны иметь возможность отправлять команды GET, POST, PUT и DELETE, что значительно повышает ясность того, что делает запрос.

Как правило, четыре основных HTTP-глагола используются следующим образом:

GET

Прочитать конкретный ресурс (по идентификатору) или набор ресурсов

PUT

Обновить конкретный ресурс (по идентификатору) или набор ресурсов. Также может использоваться для создания определенного ресурса, если идентификатор ресурса известен заранее

DELETE

Удалить конкретный ресурс по идентификатору

POST

Создать новый ресурс. Также универсальное действие для операций, которые не вписываются в другие категории

Примечание

GET-запросы не должны изменять данные базовых ресурсов. При этом может выполняться отслеживание, приводящее к обновлению данных, но данные ресурса, идентифицированного данным URI, не должны изменяться.

Давайте ресурсам продуманные имена

Создание хорошего API — это на 80% искусство и на 20% наука. Создание иерархии осмысленных URL-адресов относится к искусству. Рациональное наименование ресурсов (названия которых представляют собой просто URL-пути, такие как /customers/12345/orders) улучшает понимание того, что делает данный запрос.

Подходящие названия ресурсов предоставляют контекст для запроса и делают API сервиса более понятным. Ресурсы должны просматриваться иерархически по их именам. Пользователям должна предлагаться удобная, легко понимаемая иерархия ресурсов для использования в их приложениях.

Вот несколько простых правил для дизайна URL-пути (имени ресурса):

- Используйте идентификаторы в URL-адресах, а не в строке запроса. Использование параметров строки запроса отлично подходит для фильтрации, но не для имен ресурсов
    - Хорошо: /users/12345

    - Плохо: /api?type=user&id=23

- URL-адреса иерархичны, пользуйтесь этим для задания структуры ресурсов
- Дизайн сервиса должен быть ориентирован на ваших клиентов, а не на ваши данные
- Имена ресурсов должны быть существительными. Избегайте глаголов в именах ресурсов, это позволит сделать их яснее. Используйте методы HTTP, чтобы указать, какое действие выполняет запрос
- Используйте множественное число в соответствующих сегментах URL-адресов, чтобы обеспечить согласованность URI вашего API во всех HTTP-методах, применяя метафору коллекции
    - Хорошо: /customers/33245/orders/8769/lineitems/1
    - Плохо: /customer/33245/order/8769/lineitem/1
- Избегайте использования наборов слов в URL-адресах (например, customer_list в качестве ресурса). Используйте множественное число для названий коллекций
    - Хорошо: /customers
    - Плохо: /customer_list
- Используйте строчные буквы в URL, разделяя слова подчеркиванием («_») или дефисом (»-«). Некоторые серверы игнорируют регистр, поэтому лучше четко придерживаться нижнего регистра
- Старайтесь, чтобы URL-адреса были как можно короче и содержали как можно меньше сегментов

Используйте коды HTTP-ответов для указания статуса

Коды ответа являются частью спецификации HTTP. Для описания самых распространенных ситуаций существует большой набор HTTP-ответов.

Поскольку наши RESTful сервисы следуют спецификации HTTP, наши веб-API должны возвращать коды состояний HTTP. Например, когда ресурс успешно создан с помощью запроса POST, API должен вернуть код состояния HTTP 201. Полный список возможных кодов состояния HTTP с подробным описанием доступен здесь

Top 10 кодов состояния HTTP-ответа:

200 OK

Код, указывающий на успешное выполнение запроса и чаще всего встречающийся на практике

201 CREATED

Ресурс успешно создан (через POST или PUT). Установите заголовок Location со ссылкой на вновь созданный ресурс (при POST). Тело ответа может быть как пустым, так и содержать что-то 204 NO CONTENT Запрос выполнен успешно, но в теле ответа нет данных. Часто используется для операций DELETE и PUT

400 BAD REQUEST

Общая ошибка, когда при выполнении запроса возникает недопустимое состояние. Примеры - ошибки проверки домена, отсутствующие данные и т.д.

401 UNAUTHORIZED

Код ошибки для отсутствующего или недопустимого токена аутентификации

403 FORBIDDEN

Код ошибки, когда пользователь не авторизован для выполнения операции или ресурс недоступен по какой-либо причине (например, ограничения по времени и т.п.)

404 NOT FOUND

Этот код используется, когда запрошенный ресурс не найден. Ресурс не существует, либо была ошибка 401 или 403, которую по соображениям безопасности сервис хочет скрыть

405 METHOD NOT ALLOWED

Используется для указания на то, что запрошенный URL-адрес существует, но используемый HTTP-метод неприменим. Например, POST /users/12345, где API не поддерживает создание ресурсов таким образом (с предоставленным идентификатором). При возврате ошибки 405 должен быть установлен HTTP-заголовок Allow, указывающий на поддерживаемые методы HTTP. В примере выше заголовок выглядел бы как “Allow: GET, PUT, DELETE”

409 CONFLICT

Этот код ошибки отправляется всякий раз, когда выполнение запроса может привести к конфликту ресурсов. Примеры таких ситуаций - двойные записи, например, попытка создать двух клиентов с одинаковой информацией; удаление корневых объектов, когда не поддерживается каскадное удаление

500 INTERNAL SERVER ERROR

Никогда не отправляйте этот код вручную. Это общая ошибка, когда на стороне сервера выбрасывается какое-то исключение. Этот код должен использоваться только для ошибок, которые пользователь не может устранить со своей стороны

XML и JSON

Если вы не работаете в строго стандартизированной и регулируемой отрасли, лучше поддерживать JSON. Но если вас ничто не сковывает, позвольте пользователям выбирать в каком формате получать данные — JSON или XML. У пользователей должна быть возможность переключаться между ними с помощью HTTP-заголовка Accept или просто изменив расширение с .xml на .json.

Имейте в виду, что как только мы начинаем говорить о поддержке XML, мы начинаем говорить о валидации, пространствах имен и т.д. Если этого не требует ваша отрасль, избегайте поддержки всех этих усложнений. По крайней мере, вначале. А если в этом функционале нет острой необходимости, то всегда. JSON является простым, лаконичным и функциональным. Сделайте так, чтобы ваш XML выглядел так же, если это возможно.

Другими словами, сделайте возвращаемый XML более похожим на JSON — простым и легко читаемым, без сведений о схеме и пространстве имен, содержащим только данные и ссылки. Если ваш XML будет более сложным, стоимость поддержки будет неоправданно большой. Если судить по нашему опыту — никто никогда не отвечает в формате XML. Обрабатывать XML слишком затратно.

Обратите внимание, что JSON-Schema предлагает возможности по валидации XML, если вам все-таки нужен такой функционал.

Создавайте детальные ресурсы

Сначала гораздо проще создавать API, которые имитируют основной домен приложения или архитектуру базы данных вашей системы. В конце концов, вы захотите объединить сервисы, которые используют несколько основных ресурсов, чтобы избежать избыточности информации. Позже будет гораздо проще создать большие ресурсы из отдельных ресурсов, чем детальные ресурсы из более крупных составных ресурсов. Упростите себе задачу и начните с небольших, легко определяемых ресурсов, предоставив для них CRUD-функциональность. Ресурсы без лишней информации, ориентированные на конкретные ситуации, можно сделать позже.

Учитывайте связность

Одним из принципов REST является связность через ссылки. Хотя сервисы остаются полезными и без них, API становится более самоописательным, когда в ответе содержатся ссылки. По крайней мере, ссылка “на себя” информирует клиентов, как данные были или могут быть получены. Кроме того, используйте заголовок Location, который должен содержать ссылку на создание ресурса с помощью POST (или PUT). Для коллекций возвращайте в ответе сведения о том, что поддерживается пагинация, а также, как минимум, ссылки “первая”, “последняя”, “следующая” и “предыдущая”.

Что касается форматов ссылок, то их существует довольно много.

Спецификация HTTP веб-ссылок RFC5988 определяет ссылку следующим образом:

Ссылка — это типизированное соединение между двумя ресурсами, идентифицируемыми

интернационализированными идентификаторами ресурсов (IRI) [RFC3987]

Ссылка состоит из:

- контекстного IRI
- типа ссылки
- целевого IRI
- целевых атрибутов (опционально)

Ссылку можно рассматривать как утверждение вида {контекстный IRI} имеет ресурс {типа} в {целевом IRI}, который имеет {целевые атрибуты}.

По меньшей мере, размещайте ссылки в HTTP-заголовке Link, как это рекомендовано в спецификации, или используйте JSON-представление данного стиля HTTP-ссылок (например, ссылки в стиле Atom, см. RFC4287). По мере того, как ваш API будет становиться более зрелым, вы сможете использовать более сложные стили ссылок, такие как HAL+JSON, Siren, Collection+JSON и/или JSON-LD и т.д.

Именование ресурсов

Кроме правильного использования HTTP глаголов, именование ресурсов, вероятно, самая обсуждаемая и важная концепция для понимания во время создания понятного и легко используемого API для Web-сервиса. Когда ресурсы названы хорошо, API интуитивен и лёгок в использовании. Если же ресурсы названы плохо, тот же самый API может показаться неуклюжим и трудным в понимании и использовании. Ниже приведены несколько подсказок, как продолжить создавать URI ресурсов для нового API.

Фактически RESTful API - это всего лишь набор URI, HTTP вызовов к этим URI и некоторое количество представлений ресурсов в формате JSON и/или XML, многие из которых будут содержать перекрестные ссылки. За основу адресации берется покрытие уникальными идентификаторами ресурсов (URI) У каждого ресурса есть свой адрес или URI: вся интересная информация, которую сервер может предоставить, представлена как ресурс. Ограничение однообразия интерфейса частично реализовано с помощью комбинаций URI и HTTP глаголов и их использованием в соответствии со стандартами и конвенциями.

Когда вы решаете, какие ресурсы будут в вашей системе, называйте их существительными, в противоположность глаголам, или действиям. Другими словами, URI должен ссылаться на ресурс, а не на действие. Ещё один отличающий фактор - у существительных есть такие свойства, которых нет у глаголов.

Ниже приведены примеры ресурсов:

- Пользователи системы.
- Курсы, в которых зарегистрирован студент.
- История сообщений пользователя.
- Пользователь, который подписан на другого пользователя.
- Пользователи, которые подписаны на другого пользователя.
- Статья о верховой езде.

Каждый ресурс сервиса должен иметь хотя бы один URI, идентифицирующий его. И лучше всего, когда этот URI имеет смысл и адекватно описывает этот ресурс. URI должны иметь предсказуемую, иерархическую структуру, чтобы увеличить понятность и, как следствие, юзабилити: предсказуемость означает, что они консистентны, иерархичность означает, что у данных есть структура взаимоотношений. Это не принцип и не ограничение REST, но это улучшает API.

RESTful API пишут для потребителей. Названия и структура URI должна передавать смысл этим потребителям. Очень часто трудно понять, где должны быть границы, но с пониманием ваших данных вы поймете и то, что имеет смысл возвращать как представление вашим клиентам. Проектируйте для клиентов, а не для ваших данных.

Давайте предположим, что мы описываем систему с покупателями, заказами, отдельными позициями, продуктами и т. д. Рассмотрим URI, включенные в описание ресурсов этого сервиса:

Примеры

body_new { font-family: system-ui, -apple-system, "Segoe UI", Roboto, "Helvetica Neue", "Noto Sans", "Liberation Sans", Arial, sans-serif, "Apple Color Emoji", "Segoe UI Emoji", "Segoe UI Symbol", "Noto Color Emoji"; font-size: 1rem; font-weight: 400; line-height: 1.5; margin: 0; color: #212529; background-color: #fff; -webkit-text-size-adjust: 100%; -webkit-tap-highlight-color: transparent; } .container_new { margin: 1rem; padding: 1.5rem; border: 1px solid #dee2e6; border-radius: 0.375rem; } .tab_new { display: flex; flex-direction: column; } .tab-nav_new > input[type="radio"] { display: none; } .tab-content_new { display: none; } #content-1:has(~ .tab-nav_new > #tab-btn-1:checked), #content-2:has(~ .tab-nav_new > #tab-btn-2:checked), #content-3:has(~ .tab-nav_new > #tab-btn-3:checked) { display: block; } .tab-nav_new { display: flex; flex-wrap: wrap; border-bottom: 1px solid #dee2e6; margin-bottom: 1rem; order: -1; } .tab-nav_new > label { display: block; padding: 0.5rem 1rem; color: #0d6efd; text-decoration: none; background: 0 0; border: 1px solid transparent; margin-bottom: -1px; border-top-left-radius: 0.375rem; border-top-right-radius: 0.375rem; -webkit-appearance: button; transition: color .15s ease-in-out,background-color .15s ease-in-out,border-color .15s ease-in-out; } .tab-nav_new > input[type="radio"]:checked + label { color: #000; background-color: #fff; border-color: #dee2e6 #dee2e6 #fff; cursor: default; }

Чтобы создать нового покупателя в системе мы используем: POST http://www.example.com/customers Чтобы получить информацию о покупателе с ID# 33245: GET http://www.example.com/customers/33245 Тот же URI мы используем для PUT и DELETE, чтобы обновлять и удалять, соответственно. Ниже предложены URI для продуктов: POST http://www.example.com/products для создания нового продукта. GET|PUT|DELETE http://www.example.com/products/66432 для чтения, обновления, удаления продукта с ID# 66432, соответственно. Теперь становится весело… Как насчёт создания нового заказа у покупателя? Один вариант - POST http://www.example.com/orders. Это может работать для создания заказа, но здесь, пожалуй, не учитывается покупатель. Поскольку мы хотим создать заказ для покупателя, (заметьте связь), этот URI, очевидно, не так интуитивен, как мог бы быть. Очевидно, что следующий URI предлагает большую ясность: POST http://www.example.com/customers/33245/orders Теперь мы знаем, что создаём заказ для покупателя с ID# 33245. Что же вернет следующее? GET http://www.example.com/customers/33245/orders Вероятно, список заказов покупателя #33245. Заметьте: мы можем не поддерживать DELETE или PUT для этого URL, поскольку он оперирует коллекцией. Теперь, продолжая концепцию иерархичности, как насчёт следующего URI? POST http://www.example.com/customers/33245/orders/8769/lineitems Это может добавлять отдельную позицию в заказ #8769 (который принадлежит покупателю #33245). Точно! GET на этот URI

вернет все отдельные позиции данного заказа. Как бы то ни было, если отдельные позиции нельзя рассматривать только в контексте покупателя, или их можно рассматривать вне его контекста, мы можем предложить POST www.example.com/orders/8769/lineitems. Наряду с этими строками, поскольку может быть несколько URI для заданного ресурса, мы также можем предложить GET http://www.example.com/orders/8769, который возвращает информацию о заказе по его ID без указания ID покупателя Спускаясь глубже по иерархии: GET http://www.example.com/customers/33245/orders/8769/lineitems/1 Может возвращать только первую отдельную позицию в заказе. К этому моменту вы уже можете видеть, как работает концепция иерархичности. Нет никаких жёстких правил, убедитесь только, что предложенная структура понятна потребителю ваших сервисов. Как и все в ремесле разработки ПО, именование критично для успеха. Взгляните на некоторые широко используемые API, чтобы приобрести навык проектирования и используйте интуицию своих коллег, чтобы улучшить URI ресурсов вашего API. Ниже приведены некоторые примеры API: * Twitter: https://dev.twitter.com/docs/api * Facebook: http://developers.facebook.com/docs/reference/api/ * LinkedIn: https://developer.linkedin.com/apis

Пока мы обсуждали некоторые примеры подходящих названий для ресурсов, иногда более информативно увидеть некоторые анти-паттерны. Ниже представлены плохие примеры RESTful URI ресурсов, которые были замечены в "дикой природе." Никогда так не делайте. Прежде всего, часто сервисы используют один URI, чтобы определить интерфейс, используя строковые параметры jquery для определения операции, которую нужно выполнить и/или HTTP глагол. Например, чтобы обновить данные покупателя с ID 12345 и получить их в формате JSON может быть использован такой запрос: GET http://api.example.com/services?op=update_customer&id=12345&format=json Теперь вы не станете так делать. Несмотря на то, что узел URL-адреса 'services' — существительное, он не является самодокументируемым, поскольку иерархия URI не одна и та же для всех запросов. Кроме того, он использует глагол GET, хотя выполняет обновление. Это контринтуитивно, опасно и вызывает много боли у клиентов. Вот другой пример, который тоже выполняет обновление данных о покупателе: GET http://api.example.com/update_customer/12345 И его злобный двойник: GET http://api.example.com/customers/12345/update Вы часто можете увидеть последний запрос в сервисах других разработчиков. Заметим, что разработчики стараются создавать RESTful ресурсы и кое-где заметен прогресс. Но лучше, когда вы можете увидеть глагол в URL. Заметьте также, что нам не нужно использовать фразу 'update' в URL, потому что мы можем довериться HTTP глаголу, чтобы сообщить об этой операции. Проясним, что следующий URL избыточен: PUT http://api.example.com/customers/12345/update И с PUT, и с 'update' в запросе, мы рискуем запутать потребителей нашего сервиса. Является ли 'update' ресурсом? Мы потратим некоторое время на выяснение этого. Я уверен, вы понимаете. Давайте поговорим о споре между плюрализаторами и "сингуляризаторами". Вы не слышали об этом споре? Он существует. Соответственно, всё сводится в следующему вопросу: Должны ли узлы URI в иерархии называться существительными в единственном или множественном числе? Например, как должен выглядеть URI для получения данных о покупателе: GET http://www.example.com/customer/33245 или GET http://www.example.com/customers/33245 С обеих точек зрения приведены хорошие аргументы, но общепринятой практикой является плюрализация узлов, чтобы обеспечить согласованность для всех HTTP методов. Этот подход обоснован тем, что покупатели считаются коллекцией внутри сервиса и ID (например, 33245) ссылается на одного покупателя из коллекции. Используя это правило, получим следующие URI с множеством узлов (выделено мной): GET http://www.example.com/customers/33245/orders/8769/lineitems/1 ноды 'customers', 'orders', и 'lineitems' использованы во множественном числе. Это подразумевает, что вам нужно только два базовых URL для каждой корневой сущности. Один для создания ресурса внутри коллекции и другой для чтения, обновления и удаления ресурса по его идентификатору.

Например, для создания (используя покупателей) POST http://www.example.com/customers A для чтения, обновления и удаления следующий: GET|PUT|DELETE http://www.example.com/customers/{id} Как было замечено ранее, у ресурса может быть несколько URI, но для минимальных возможностей CRUD'a достаточно всего двух. Вы можете спросить: есть ли ситуации, когда плюрализация не имеет смысла. Да, фактически есть. Когда в системе не требуется коллекция. Другими словами, приемлем singleton ресурс. Например, если есть единственный всеобъемлющий конфигурационный ресурс, вы можете использовать существительное в единственном числе, чтобы это отразить: GET|PUT|DELETE http://www.example.com/configuration Обратите внимание на отсутствие ID и использования глагола POST. Вы скажете, что если бы у каждого покупателя могла быть только одна конфигурация, тогда URL мог бы быть таким: GET|PUT|DELETE http://www.example.com/customers/12345/configuration Снова нет никакого ID для конфигурации и нет использования глагола POST. Хотя я уверен, что в обоих случаях можно утверждать, что использование POST является допустимым. Ладно, хорошо.
○ Примеры URI ресурсов ○ Плохие примеры именования ресурсов ○ Множественность