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