ischerbin карма 33,0 41 голос рейтинг 0,0

22 мая 2012 в 11:36

RESTful API для сервера – делаем правильно (Часть 2) перевод

Мобильный веб*, Веб-разработка*

В первой части статьи я кратко описал принципы RESTful и объяснил каким образом следует проектировать архитектуру вашего сервера так, чтобы можно было легко выпускать новые и прекращать поддержку устаревших версий вашего API. В этой части я кратко расскажу о HATEOAS и Hypermedia, а затем расскажу о роли, которую они могут сыграть при разработке нативных приложений для мобильных устройств. Но главной темой этой статьи будет реализация кэширования (точнее поддержка кэширования на стороне сервера). Целевая аудитория включает разработчиков серверного ПО и, в какой то мере, разработчиков под iOS или под другие мобильные платформы.

HTTP API, REST и HATEOAS

В настоящее время HTTP API можно разделить на

- Web Services
- RPC URI Tunnelling
- HTTP-based Type 1
- HTTP-based Type 2
- REST

Здесь находится очень хорошее объяснение этого от Jon Algermissen. К сожалению каждый, кто предоставляет свой API называет свой сервис RESTful даже несмотря на то, что он таковым не является.
Так что же это такое, настоящий RESTful сервер? Это любой API, основанный на hypertext/hypermedia. Другими словами, сторонний разработчик или клиентское приложение должны иметь возможность получить информацию о “других доступных ресурсах” через корневой URL API. На самом деле это самое важное условие при реализации RESTful API. Кроме того, настоящим RESTful сервером может считаться только тот, который придерживается принципа HATEOAS.

HATEOAS – Hypermedia as the Engine of Application State

Два основных принципа, которым необходимо следовать при реализации HATEOAS гласят:

1. **Обслуживайте только гиперссылочные (hypermedia) ресурсы.** Гиперссылочные ресурсы это такие, которые содержат только информационное наполнение и ссылки (hyperlinks) на другие гиперссылочные ресурсы. JSON (application/json) НЕ является гиперссылочным ресурсом. (С другой стороны, существует множество RESTful, HATEOAS серверов, которые поддерживают JSON) Однако вы можете добавить дополнительные поля в JSON, заставив его таким образом действовать как гиперссылочный ресурс. (Например: поле href для ссылки на соответствующую превьюшку картинки)
2. **Единая точка входа для клиентского приложения.** С домашней страницы API последующие GET вызовы должны быть оформлены как ссылки на соответствующие URL, а последующие “POST”, “PUT” или “DELETE” запросы в виде форм.

Главное преимущество, которое вы получаете следуя этим двум принципам HATEOAS, это простота документирования.

Использовать ли HATEOAS в вашем новом API?

НЕТ!
Почему? В прошлом API в основном писались для использования в web приложениях. Эти сервера обычно возвращали XHTML, а клиентские приложения выполнялись в браузере. Браузер в такой схеме является подобием вашего мобильного клиента, который парсит гиперссылочные ресурсы, знает что такое форма и как представить ее пользователю. В случае с мобильным приложением, когда ответ вашего RESTful сервера содержит форму, данные из которой вы можете отправить на сервер, вы не сможете (по крайней мере без дополнительных усилий) конвертировать ее в нативные элементы интерфейса, доступные на платформе. Никто из поставщиков платформ, ни Apple/Google или Microsoft, не предоставляет поддержку конвертирования XHTML форм в UIViewController (на iOS) или в Intent (на Android) или в Silverlight Page (на Windows Phone). Я рекомендую использовать выдачу гиперссылочных ресурсов в ответ на GET запросы и предоставлять вызовы методов контроллера (вместо форм) для всех POST/PUT и DELETE запросов. (Например: /friends/add или /venues/checkin) Вызовы методов контроллера могут быть встроены в другие ответы (для того, чтобы клиентское приложение или разработчик могли узнать о них). Конечно это нарушает принципы REST, но ничего страшного. Лучше делать качественный продукт, чем слепо

следовать стандартам, от которых на практике толку мало. (Это возможно приведет к тому, что наш API станет HTTP-based Type 2).

Кэширование

Переходим к кэшированию. Кэширование, как многие считают, в основном клиентская задача (или задача промежуточного прокси). Но вы знаете что, ценой небольших усилий при разработке серверной части, вы можете сделать ваш API полностью отвечающим требованиям промежуточных, кэширующих прокси? Это значит, что вы получите бесплатную балансировку нагрузки с их стороны. Все что нужно описано в 13-й главе спецификации HTTP.

Не так давно Арт Тейлор написал в своем твиттере:

"Я вывел два новых правила: 1) Если ваше приложение тормозит — добавьте кэширование. 2) Если приложение глючит — уберите кэширование. Ну почему кэширование так сложно!"

Но поверьте, кэширование может быть реализовано без столкновений с такими проблемами. Два главных принципа, которым я вам рекомендую следовать это:

1. Не пытайтесь делать нестандартные схемы кэширования в клиентском приложении.
2. Разберитесь с базовыми принципами кэширования, описанными в RFC спецификации HTTP 1.1. Там описаны две модели кеширования. Модель срока действия и модель действительности (валидности).

В любом клиент-серверном приложении сервер — заслуживающий доверия источник информации. Когда вы загружаете ресурс (страницу или ответ) с API сервера, сервер отправляет клиенту, кроме всего прочего, некоторые дополнительные "подсказки" как клиент может кэшировать полученный ресурс. Сервер авторитетно указывает клиенту когда срок действия кэшированной информации истекает. Эти подсказки могут быть отправлены как программно, так и через настройку сервера. Модель срока действия обычно реализуется через настройку конфигурации сервера, в то время как модель валидности требует программной реализации силами разработчика серверной части. Именно разработчик должен решить когда использовать валидность, а когда срок действия исходя из типа возвращаемого ресурса. Модель срока действия обычно используется когда сервер может однозначно определить как долго тот или иной ресурс будет действительным. Модель валидности используется для всех остальных случаев. Позже я покажу вам как реализовать обе эти модели в ходе разработки сервера. Как только вы разберетесь с обеими, я покажу когда использовать каждую из них.

Модель срока действия

Давайте рассмотрим распространенную конфигурацию кэширования. Если вы используете nginx, у вас наверняка есть в конфиге нечто подобное:

```
location ~ \.(jpg|gif|png|ico|jpeg|css|swf)$ {
                expires 7d;
        }
```

nginx переводит эти настройки в соответствующий заголовок HTTP. В данном случае сервер отправляет поле "Expires" или "Cache-Control: max-age=n" в заголовке для всех изображений и рассчитывает на то, что клиент закэширует их на 7 дней. Это значит, что вам не нужно будет запрашивать эти же данные в течение 7-ми последующих дней. Каждый из распространенных браузеров (и промежуточных прокси) учитывает этот заголовок и работает как ожидается. К сожалению большинство Open Source фреймворков кэширования изображений для iOS, включая популярный SDWebImage, используют встроенный механизм кэширования, просто удаляющий изображения после n дней. Проблема заключается в том, что такие фреймворки не соответствуют модели валидности и ваше клиентское приложение, использующее эти фреймворки вынуждено прибегать к нестандартным решениям (хакам). Я приведу пример, показывающий, что тут может пойти не так. Вернемся к нашему "новому Фейсбуку". Когда ваш пользователь загружает на сервер аватарку, он считает что изменения отразятся во всех представлениях. Некоторые хитрые разработчики очищают локальный кэш после успешного вызова update-profile-image. (Это значит что все контроллеры должны загрузить картинку с сервера по новой). Все работает замечательно, вы отчитались перед менеджером проекта и в каждом представлении теперь отображается самая свежая картинка из профиля. Однако полностью проблему это не решает. Новую аватарку пользователя его друзья увидят только через 7 дней. Абсолютно неприемлемо. Так как это решить? Как я уже сказал, вы должны принять утверждение, что только сервер может быть источником достоверных данных. Не используйте нечестные трюки на клиенте для обновления кэша путем преждевременного окончания срока действия кэшированного контента.

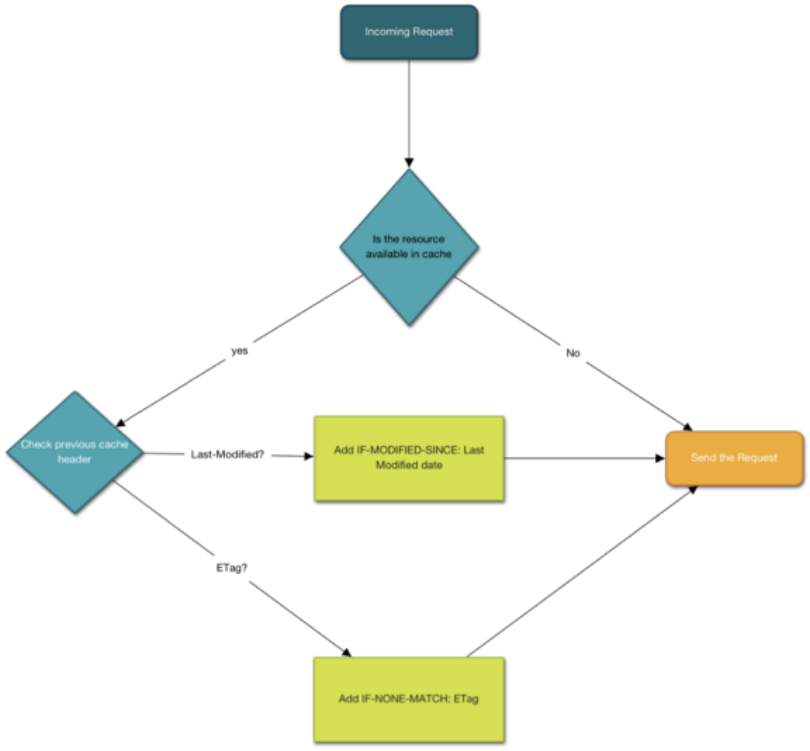
Модель валидности

И Facebook и Twitter решают проблему устаревших изображений в профиле (после того как было загружено новое изображение) используя модель валидности. В модели валидности сервер отправляет клиенту уникальный идентификатор ресурса и клиент кэширует и идентификатор и ответ. В терминах HTTP такой уникальный

идентификатор называется ETag. Когда вы совершаете второй запрос к тому же ресурсу, вы должны отправить его ETag. Сервер использует этот идентификатор для проверки был ли изменен запрашиваемый вами ресурс с момента последнего обращения (помните, сервер — единственный достоверный источник). Если ресурс действительно менялся он отправляет последнюю копию. В противном случае он шлет 304 Not Modified. Модель валидности кэша требует для реализации дополнительных усилий от разработчика при разработке как клиентской, так и серверной частей. Я опишу их обе далее.

Поддержка на стороне клиента

На самом деле под iOS, если вы используете MKNetworkKit он делает всю работу автоматически. Но для разработчиков под Android и Windows Phone я распишу подробно как это следует реализовывать.
Модель валидности кэша использует ETag и Last-Modoified заголовки HTTP. Реализация клиентской части проще чем серверной. Если вы получили ETag с ресурсом, когда вы делаете второй запрос на получение его же, отправьте ETag в поле “IF-NONE-MATCH” заголовка. Аналогично, если вы получили “Last-Modified” с ресурсом, отправьте его в поле “IF-MODOFIED-SINCE” заголовка в последующих запросах. Сервер же со своей стороны сам решит когда использовать “ETag”, а когда “Last-Modified”.



Реализация модели срока действия проста. Просто рассчитайте дату окончания срока действия на основе полей заголовка, “Expires” или “Cache-Control: max-age-n” и очистите кэш при наступлении этой даты.

Реализация на стороне сервера

Использование ETag
ETag обычно рассчитывается на сервере с использованием алгоритмов хэширования. (Большинство серверных языков высокого уровня, таких как Java/C#/Scala обладают средствами хэширования объектов). Перед формированием ответа сервер должен рассчитать хэш объекта и добавить его в поле заголовка ETag. Теперь, если клиент действительно отправил IF-NONE-MATCH в запросе и данный ETag равен тому, что вы рассчитали, отправьте 304 Not Modified. Иначе сформируйте ответ и отправьте его с новым ETag.

Использование Last-Modified
Реализация использования Last-Modified не совсем проста. Давайте представим что в нашем API есть вызов, возвращающий список друзей.

```
http://api.mynextfacebook.com/friends/
```

Когда вы используете ETag, вы вычисляете хэш массива друзей. При использовании Last-Modified вы должны отправлять дату последнего изменения этого ресурса. Поскольку этот ресурс представляет собой список, эта дата должна являть собой дату когда вы последний раз добавили нового друга. Это требует от разработчика организации хранения даты последнего изменения данных для каждого пользователя в базе. Немного сложнее чем ETag, но дает большое преимущество в плане производительности.
Когда клиент запрашивает ресурс первый раз, вы отправляете полный список друзей. Последующие запросы от

клиента теперь будут иметь поле “IF-MODIFIED-SINCE” в заголовке. Ваш серверный код должен отправлять только список друзей, добавленных после указанной даты. Код обращения к базе до модификации был примерно таким:

```
SELECT * FROM Friends;
```

после модификации стал таким:

```
SELECT * FROM Friends WHERE friendedDate > IF-MODIFIED-SINCE;
```

Если запрос не вернет записей, отправляем 304 Not Modified. Таким образом, если у пользователя 300 друзей и только двое из них были добавлены недавно, то ответ будет содержать только две записи. Время обработки запроса сервером и затрачиваемые при этом ресурсы снижаются значительно.
Конечно это сильно упрощенный код. Разработчику добавится головной боли когда вы решите сделать поддержку удаления или блокирования друзей. Сервер должен быть способным отправлять подсказки, используя которые у клиента будет возможность сказать какие друзья были добавлены, а какие удалены. Эта техника требует дополнительных усилий при разработке серверной части.

Выбор модели кэширования

Итак. Это была непростая тема. Теперь я попробую подвести итоги и вывести базовые правила использования той или иной модели кэширования.

1. Все статические изображения должны обслуживаться по модели срока действия.
2. Все данные, формируемые динамически, должны кэшироваться по модели валидности.
3. Если ваш, динамически формируемый, ресурс является списком, вам следует использовать модель валидности, основанную на Last-Modified. (Например /friends). В остальных случаях следует использовать модель валидности, основанную на ETag. (Например /friends/firstname.lastname).
4. Изображения или любые другие ресурсы, которые могут быть изменены пользователем (такие как аватар) должны также кэшироваться по модели валидности с использованием ETag. Несмотря на то, что это изображения, они не постоянны как например логотип компании. Кроме того вы просто не сможете точно рассчитать срок действия таких ресурсов.

Другой способ (более простой в реализации, но немного хакерский), это использование “ошибки URL”. Когда в ответе есть URL аватара, надо сделать часть его динамичной. Так вместо представления URL как

```
http://images.mynextfacebook.com/person/firstname.lastname/avatar
```

сделать

```
http://images.mynextfacebook.com/person/firstname.lastname/avatar/<хэш>
```

Хэш должен меняться в случае когда пользователь меняет аватар. Вызов, отправляющий список друзей, теперь отправит модифицированные URL-ы для пользователей, сменивших свои аватары. Таким образом изменения в изображениях профиля будут распространяться практически моментально!
Если ваши серверные и клиентские приложения будут соответствовать практически устоявшимся стандартам кэширования, ваше iOS приложение и ваш продукт вообще будут просто “летать”.

В этой статье я дал простое объяснение таких стандартов, которых подавляющее большинство разработчиков не придерживаются.

На этом я заканчиваю вторую часть статьи. Следующая и последняя будет описывать обмен информацией об ошибках и их правильную обработку, а также интернационализацию вашего приложения.

Рекомендую к прочтению

REST API Design Rulebook

restful, web-services

+21 29623 392 Mugunth Kumar ischerbin 0,0

Комментарии (14)

So1 **So1** 22 мая 2012 в 12:25 # 0

Это — хорошее пособие для новичков, на деле «просто» все-равно не получится, потому что инвалидация кеша в некоторых ситуациях сопровождается баттхертом. Вы привели отличную цитату Арта Тейлора, я приведу другую:

There are only two hard things in Computer Science: cache invalidation and naming things.

— Phil Karlton

за эту цитату спасибо (habrahabr.ru/users/zerkms/, если хабр порежет юзер-тэг, — не знаю с чем связано, у меня Хабр частенько тэги режет)

forefinger 22 мая 2012 в 16:03 # +1

Простите, я что-то не понял:
>Если запрос не вернет записей, отправляем 304 Not Modified. Таким образом, если у пользователя 300 друзей и только двое из них были добавлены недавно, то ответ будет содержать только две записи.

Насколько я понимаю, если есть изменения после IF-MODIFIED-SINCE — нужно прислать обновленный объект (список друзей), а не разницу. Так как решается вопрос инвалидации имеющегося у клиента кэша, таким образом, если объект на сервере изменен — это не проблемы клиента как именно он изменился, ему нужен актуальный объект. Или здесь какая-то хитрая RESTful API собака зарыта?

ischerbin 22 мая 2012 в 16:39 # 0

Если на клиенте есть список из 298 друзей, то докачиваем 2 недостающих. Видимо так.

forefinger 22 мая 2012 в 18:12 # 0

Я уточню вопрос — представим себе абстрактное клиентское приложение делающее запросы:

```
No cache version: запрос http://api.mynextfacebook.com/friends/ -> 200 OK -> 298 друзей
IF-MODIFIED-SINCE=10: запрос http://api.mynextfacebook.com/friends/ -> 304 Not Modified
... произошло добавление двух друзей ...
IF-MODIFIED-SINCE=20: запрос http://api.mynextfacebook.com/friends/ -> 200 OK -> 2 друга
IF-MODIFIED-SINCE=30: запрос http://api.mynextfacebook.com/friends/ -> 304 Not Modified
```

Цифры 10, 20, 30 — взяты для примера, думаю что всем понятно что там должен быть честный timestamp.

Получая такие ответы от сервера приложение должно merge имеющего кэша и нового ответа, результат этого merge оно должно закэшировать и именного его использовать. Но при этом отличить ответ — это полный объект или только обновление при кэшировании можно только по заголовкам.

Но при использовании какого-нибудь network framework'а приложение ожидая ответ от сервера на запрос списка друзей получает совершенно не то:
— приложение может и знать-не-знать про кэширование — так как эту задачу на себя берет framework, следовательно оно не обязано знать что до такого-то момента у пользователя было 298 друзей;
— network framework — совершенно не обязан делать merge кэша и результата нового запроса, так как его задача получить данные, а что они из себя представляют JSON/XML/XHTML — это не его головная боль, следовательно не его дело менять данные ответа от сервера.

Таким образом получается что для работы такого поведения сервера клиентское приложение должно хранить 2 версии кэша — network framework, для того чтобы работать по rfc2616, + приложение должно знать о том какое было состояние до запроса, чтобы вычислить текущее положение дел (сделать merge). А если добавить к этому то что приложение вынужденно знать использовался ли кэш или нет (чтобы понять как обрабатывать ответ — добавлять 2 друзей или список состоит из 2-х друзей) — возникает вопрос network framework получается нужен чтобы сокет открыть и пописать+почитать в/из него? Network framework используют как-раз с целью избавить приложение от необходимости работать с сетью. И как cherry-on-top вспомним то, что кэш — субстанция которая может быть удалена разными механизмами и далеко не все из них удосужатся сообщить приложению об удалении его кэша.

Таким образо либо я что-то не понял из статьи, либо что-то очень хитрое придумал, либо тут все-таки RESTful API собака зарыта.

P.S. под network framework — понимаются например MKNetworkKit или иной, которому для работы нужно лишь сказать куда класть кэш и можно начинать делать запросы.

T2L 22 мая 2012 в 23:58 # 0

Когда есть `IF-MODIFIED-SINCE`, тогда присылается только разница (2 новых друга из примера).
Когда нужен полный список просто не отправляем заголовок `IF-MODIFIED-SINCE` (что в принципе тоже самое, что послать 0). API должен вернуть полный набор.

forefinger 23 мая 2012 в 13:32 # 0

Простите, но как приложение поймет что ответ является только разницей, а не полным набором, если оно использует фреймворк для работы с сетью/http, а не само рулит кэшированием? Ведь в этом случае оно не знает какие были использованы заголовки, а исходя из статьи выходит что формат ответа будет идентичен, за исключением кол-ва элементов, что не может быть достоверным признаком для определения типа ответа (обновление или полные данные).

m_z 25 мая 2012 в 11:44 # 0

Тогда автору следует ознакомиться с RFC 2616:

```
If the variant has been modified since the If-Modified-Since
date, the response is exactly the same as for a normal GET.
```

Можно было бы придумать что-то с RANGE, но как тогда быть с удаленными и измененными данными не ясно.

grossws 10 декабря 2013 в 19:20 (комментарий был изменён) # ↑ 0

Немного некрофилия, конечно.

Если хочется реализовать пересылку диффов (съэкономить на пересылке больших списков), то есть следующий подход:
— делаем два метода: `/friends` и `/friends/diff`
— оба принимают `If-Modified-Since`
— первый метод (`/friends`) при наличии изменений в списке друзей высылает обновленный список *полностью* (как и без заголовка If-Modified-Since) или отдаёт HTTP 304 (если изменений не было)
— второй метод (`/friends/diff`) при наличии изменений высылает структурированный разностный документ (с полями added и deleted, например) или отдаёт HTTP 304 (если изменений не было).

Вариант, описанный в статье, является опасной ересью (возможно, что просто в силу упрощения) и может дать сложно-диагностируемые проблемы. Например, при использовании какого-нибудь REST-framework, который сам управляет кэшированием.

BlessMaster 4 января 2014 в 23:04 (комментарий был изменён) # ↑ 0

Имхо, в постах подобного типа «некрофилия» более чем уместна, тем более с таким шикарным разруливанием достаточно распространённой ситуации. Отдельное спасибо за сам trick)

grossws 4 января 2014 в 23:16 # ↑ 0

Всегда пожалуйста =)

Сам всегда радуюсь, находя в комментариях полезные сведения. Иногда даже оказываются полезнее самой статьи.

petrkozorezov 25 мая 2012 в 11:28 # ↑ 0

IF-MODIFIED-SINCE и IF-NONE-MATCH при отсутсвии изменений присылают 304, при наличии — новый объект целиком, который в свою очередь кешируется и используется дальше

petrkozorezov 25 мая 2012 в 11:31 # 0

Для понимания и написания REST API очень хорошо подходит Webmachine, там всё по полочкам разложено.
Правда она на эрланге. Может есть аналоги и на другие языки?

roman_truschev 8 октября 2012 в 00:53 # ↑ 0

порт на рубях

roman_truschev 8 октября 2012 в 00:54 # ↑ 0

/webmachine-ruby

Только зарегистрированные пользователи могут оставлять комментарии. Войдите, пожалуйста.