
ischerbin карма 33,0 рейтинг 0,0
41 голос

17 мая 2012 в 12:04

RESTful API для сервера – делаем правильно (Часть 1) перевод

Мобильный веб*, Веб-разработка*

В 2007-м Стив Джобс представил iPhone, который произвел революцию в высокотехнологичной индустрии и изменил наш подход к работе и ведению бизнеса. Сейчас 2012-й и все больше и больше сайтов предлагают нативные iOS и Android клиенты для своих сервисов. Между тем не все стартапы обладают финансами для разработки приложений в дополнение к основному продукту. Для увеличения популярности своего продукта эти компании предлагают открытые API, которыми могут воспользоваться сторонние разработчики. Пожалуй Twitter был первым в этой сфере и теперь число компаний, последовавших этой стратегии, растет стремительно. Это действительно отличный способ создать привлекательную экосистему вокруг своего продукта.

Жизнь стартапа полна перемен, поворотных моментов, в которых от принятых решений зависит дальнейшая судьба проекта. Если ваша кодовая база не сможет обеспечить воплощение самых разных ваших решений – вы проиграли. Серверный код, который достаточно гибок для того, чтобы в короткие сроки подстроиться под нужды бизнеса, решает быть проекту или нет. Успешные стартапы не те, которые просто предложили отличную идею, но те, которые смогли ее качественно воплотить. Успех стартапа зависит от успешности его продукта, будь то приложение под iOS, сервис или API. Последние три года я работал над разными приложениями под iOS (в основном для стартапов) использовавшими web сервисы и в этом блоге я попытался собрать накопленные знания воедино и показать вам лучшие методики, которым вам нужно следовать при разработке RESTful API. Хороший RESTful API тот, который можно менять легко и просто.

Целевая аудитория

Этот пост предназначен для тех, кто обладает знаниями в разработке RESTful API уровня от средних до продвинутых. А также некоторыми базовыми знаниями объектно-ориентированного (или функционального) программирования на таких серверных языках как Java/Ruby/Scala. (Я намеренно проигнорировал PHP или Programmable Hyperlinked Pasta).
Прим. Пер. Тут автор привел ссылку на полушутливую статью о истории языков программирования где PHP был расшифрован как Programmable Hyperlinked Pasta (Программируемая Гиперссылочная Лапша). Что как бы характеризует отношение автора к PHP.

Структура и организация статьи

Статья довольно подробна и состоит из двух частей. Первая описывает основы REST тогда как вторая описывает документирование и поддержку разных версий вашего API. Первая часть для новичков, вторая для профи. Я не сомневаюсь, что вы профи, а потому вот вам ссылка чтобы перескочить сразу к главе «Документирование API». Возможно, вам стоит начать оттуда, если вам кажется, что этот пост из разряда «Многа букаф, ниасилил…».

Принципы RESTful

Сервер может считаться RESTful если он соответствует принципам REST. Когда вы разрабатываете API, который будет в основном использоваться мобильными устройствами, понимание и следование трем наиважнейшим принципам может быть весьма полезным. Причем не только при разработке API, но и при его поддержке и развитии в дальнейшем. Итак, приступим.

Независимость от состояния (Statelessness)

Первый принцип – независимость от состояния. Проще говоря, RESTful сервер не должен отслеживать, хранить и тем более использовать в работе текущую контекстную информацию о клиенте. С другой стороны клиент должен взять эту задачу на себя. Другими словами не заставляйте сервер помнить состояние мобильного устройства, использующего API.

Давайте представим, что у вас есть стартап под названием «Новый Фейсбук». Хороший пример, где разработчик мог совершить ошибку это предоставление вызова API, который позволяет мобильному устройству установить последний прочитанный элемент в потоке (назовем его лентой Фейсбука). Вызов API, обычно возвращающий ленту (назовем его /feed), теперь будет возвращать элементы, которые новее установленного. Звучит умно, не правда ли? Вы «оптимизировали» обмен данными между клиентом и сервером? А вот и нет.

Что может пойти не так в приведенном случае, так это то, что если ваш пользователь использует сервис с двух или трех устройств, то, в случае когда одно из них устанавливает последний прочитанный элемент, то остальные не смогут загрузить элементы ленты, прочитанные на других устройствах ранее.

Независимость от состояния означает, что данные, возвращаемые определенным вызовом API, не должны зависеть от вызовов, сделанных ранее.

Правильный способ оптимизации данного вызова – передача времени создания последней прочитанной записи ленты в качестве параметра вызова API, возвращающего ленту (/feed?lastFeed=20120228). Есть и другой, более «правильный» метод – использование заголовка HTTP If-Modified-Since. Но мы пока не будем углубляться в эту сторону. Мы обсудим это во второй части.

Клиент же со своей стороны, может (должен) помнить параметры, сгенерированные на сервере при обращении к нему и использовать их для последующих вызовов API, если потребуется.

Кэшируемая и многоуровневая архитектура

Второй принцип заключается в предоставлении клиенту информации о том, что ответ сервера может быть кэширован на определенный период времени и использоваться повторно без новых запросов к серверу. Этим клиентом может быть как само мобильное устройство, так и промежуточный прокси сервер. Я расскажу подробнее о кэшировании во второй части.

Клиент – серверное разделение и единый интерфейс

RESTful сервер должен прятать от клиента как можно больше деталей своей реализации. Клиенту не следует знать о том, какая СУБД используется на сервере или сколько серверов в данный момент обрабатывают запросы и прочие подобные вещи. Организация правильного разделения функций важна для масштабирования если ваш проект начнет быстро набирать популярность.

Это пожалуй три самых важных принципа, которым нужно следовать в ходе разработки RESTful сервера. Далее будут описаны три менее важных принципа, но все они имеют непосредственное отношение к тому, о чем мы тут говорим.

REST запросы и четыре HTTP метода

GET
POST
PUT
DELETE

Принцип “кэшируемости” и GET запросы

Главное, что следует помнить — вызов, совершенный через GET не должен менять состояние сервера. Это в свою очередь значит, что ваши запросы могут кэшироваться любым промежуточным прокси (снижение нагрузки). Таким образом Вы, как разработчик сервера, не должны публиковать GET методы, которые меняют данные в вашей базе данных. Это нарушает философию RESTful, особенно второй пункт, описанный выше. Ваши GET вызовы не должны даже оставлять записей в access.log или обновлять данные типа “Last logged in”. Если вы меняете данные в базе, это обязательно должны быть методы POST/PUT.

То самое обсуждение POST vs PUT

Спецификация HTTP 1.1 гласит, что PUT идемпотентен. Это значит, что клиент может выполнить множество PUT запросов по одному URI и это не приведет к созданию записей дубликатов. Операции присвоения — хороший пример идемпотентной операции

```
String userId = this.request["USER_ID"];
```

Даже если эту операцию выполнить дважды или трижды, никакого вреда не будет (кроме лишних тактов процессора). POST же с другой стороны не идемпотентен. Это что-то вроде инкремента. Вам следует использовать POST или PUT с учетом того является ли выполняемое действие идемпотентным или нет. Говоря языком программистов, если клиент знает URL объекта, который нужно создать, используйте PUT. Если клиент знает URL метода/класса создающего нужный объект, используйте POST.

```
PUT www.example.com/post/1234
```

Используйте PUT если клиент знает URI, который сам бы мог быть результатом запроса. Даже если клиент вызовет это PUT метод много раз, какого либо вреда или дублирующих записей создано не будет.

```
POST www.example.com/createpost
```

Используйте POST если сервер сам создает уникальный идентификатор и возвращает его клиенту. Дублирующие записи будут создаваться если этот запрос будет повторяться позже с такими же параметрами.
Более подробная информация в данном обсуждении.

Метод DELETE

DELETE абсолютно однозначен. Он идемпотентен как и PUT, и должен использоваться для удаления записи если таковая существует.

REST ответы

Ответы от Вашего RESTful сервера могут использовать в качестве формата XML или JSON. Лично я предпочитаю JSON, поскольку он более лаконичен и по сети передается меньший объем данных нежели при передаче такого же ответа в формате XML. Разница может быть порядка несколько сотен килобайт, но, с учетом скоростей 3G и нестабильности обмена с мобильными устройствами, эти несколько сотен килобайт могут иметь значение.

Аутентификация

Аутентификация должна производиться через https и клиент должен посылать пароль в зашифрованном виде. Процесс получения sha1 хэша NSString в Objective-C достаточно понятен и прост и приведенный код наглядно это показывает.

```
- (NSString *) sha1
{
    const char *cstr = [self cStringUsingEncoding:NSUTF8StringEncoding];
    NSData *data = [NSData dataWithBytes:cstr length:self.length];

        uint8_t digest[CC_SHA1_DIGEST_LENGTH];

        CC_SHA1(data.bytes, data.length, digest);

        NSMutableString* output = [NSMutableString stringWithCapacity:CC_SHA1_DIGEST_LENGTH * 2];

        for(int i = 0; i <; CC_SHA1_DIGEST_LENGTH; i++)
        [output appendFormat:@"%02x", digest[i]];

        return output;
}
```

Сервер должен сравнить полученный хэш пароля с сохраненным в его базе хэшем. В любом случае не следует ни при каких условиях передавать пароли с клиента на сервер в открытом виде. Из этого правила не существует исключений! День, когда Ваши пользователи узнают, что вы храните их пароли в открытом виде, может стать последним днем вашего стартапа. Доверие, потерянное однажды, вернуть невозможно.

RFC 2617 описывает два способа аутентификации на HTTP сервере. Первый — это Basic Access, второй Digest. Для мобильных клиентов подходит любой из этих двух методов и большинство серверных (и клиентских тоже) языков обладают встроенными механизмами для реализации таких схем аутентификации.

Если вы планируете сделать свой API публичным, вам следует также посмотреть в сторону oAuth или лучше oAuth 2.0. oAuth позволит Вашим пользователям публиковать контент, созданный в Вашем приложении, на других ресурсах без обмена ключами (логинами/паролями). oAuth также позволяет пользователям контролировать что именно находится в доступе и какие разрешения даны сторонним ресурсам.

Facebook Graph API это наиболее развитая и распространенная реализация oAuth на данный момент. Используя oAuth, пользователи Facebook могут давать доступ к своим фотографиям сторонним приложениям без публикации другой приватной и идентификационной информации (логин/пароль). Пользователь также может ограничить доступ нежелательным приложениям без необходимости менять свой пароль.

До сего момента я говорил об основах REST. Теперь переходим к сути статьи. В последующих главах я буду говорить о практических приемах, которые следует использовать при документировании, создании новых и завершении поддержки старых версий своего API…

Документирование API

Худшая документация, которую может написать разработчик сервера — это длинный, однообразный список вызовов API с описанием параметров и возвращаемых данных. Главная проблема такого подхода заключается в том, что внесение изменений в сервер и формат возвращаемых данных по мере развития проекта становится кошмаром. Я внесу кое какие предложения на этот счет, чтобы разработчик клиентского ПО понимал Вас лучше. Со временем это также поможет Вам в развитии в качестве разработчика серверного ПО.

Документация

Первым шагом я бы порекомендовал подумать об основных, высокоуровневых структурах данных (моделях), которыми оперирует ваше приложение. Затем подумайте над действиями, которые можно произвести над этими компонентами. Документация по foursquare API хороший пример, который стоит изучить перед тем как начать писать свою. У них есть набор высокоуровневых объектов, таких как места, пользователи и тому подобное. Также у них есть набор действий, которые можно произвести над этими объектами. Поскольку вы знаете высокоуровневые объекты и действия над ними в вашем продукте, создание структуры вызовов API становится проще и понятней. Например, для добавления нового места логично будет вызвать метод наподобие /venues/add

Документируйте все высокоуровневые объекты. Затем документируйте запросы и ответы на них, используя эти высокоуровневые объекты вместо простых типов данных. Вместо того, чтобы писать “Этот вызов возвращает три строковых поля, первое содержит id, второе имя, а третье описание” пишите “Этот вызов возвращает структуру (модель), описывающую место”.

Документирование параметров запроса

Давайте представим, что у Вас есть API, позволяющий пользователю входить, используя Facebok token. Вызовем этот метод как /login.

```
Request
/login
Headers
Authorization: Token XXXXX
User-Agent: MyGreatApp/1.0
Accept: application/json
Accept-Encoding: compress, gzip
Parameters
Encoding type – application/x-www-form-urlencoded
token – “Facebook Auth Token” (mandatory)
profileInfo = “json string containing public profile information from Facebook” (optional)
```

Где profileinfo высокоуровневый объект. Поскольку вы уже задокументировали внутреннюю структуру этого объекта то такого простого упоминания достаточно. Если Ваш сервер использует такие же Accept, Accept-Encoding и параметр Encoding type всегда вы можете задокументировать их отдельно, вместо повторения их во всех разделах.

Документирование параметров ответа

Ответы на вызовы API должны также быть задокументированы, основываясь на высокоуровневой модели объектов. Цитируя тот же пример foursquare, вызов метода /venue/#venueid# вернет структуру данных (модель), описывающую место проведения мероприятия.

Обмен идеями, документирование или информирование других разработчиков о том, что вы вернете в ответ на запрос станет проще если Вы задокументируете ваш API используя структуру объектов (моделей). Наиболее важный итог этой главы — это необходимость воспринимать документацию как контракт, который заключаете Вы, как разработчик серверной части и разработчики клиентских приложений (iOS/Android/Windows Phone/ Чтобытонибыло).

Причины создания новых и прекращения поддержки старых версий вашего API

До появления мобильных приложений, в эпоху Web 2.0 создание разных версий API не было проблемой. И клиент (JavaScript/AJAX front-end) и сервер разворачивались одновременно. Потребители (ваши клиенты) всегда использовали самую последнюю версию клиентского ПО для доступа к системе. Поскольку вы — единственная компания, разрабатывающая как клиентскую так и серверную часть, вы полностью контролируете то как используется ваш API и изменения в нем всегда сразу же применялись в клиентской части. К сожалению это невозможно с клиентскими приложениями, написанными под разные платформы. Вы можете развернуть API версии 2, считая что все будет отлично, однако это приведет к неработоспособности приложений под iOS, использующих старую версию. Поскольку еще могут быть пользователи, использующие такие приложения несмотря на то, что вы выложили обновленную версию в App Store. Некоторые компании прибегают к использованию Push уведомлений для напоминаний о необходимости обновления. Единственное к чему это приведет — потеря такого клиента. Я

видел множество айфонов, у которых было более 100 приложений, ожидающих обновления. Шансы, что ваше станет одним из них, весьма велики. Вам всегда надо быть готовым к разделению вашего API на версии и к прекращению поддержки некоторых из них как только это потребуется. Однако поддерживайте каждую версию своего API не менее трех месяцев.

Разделение на версии

Развертывание вашего серверного кода в разные папки и использование разных URL для вызовов не означает что вы удачно разделили ваш API на версии.
Так example.com/api/v1 будет использоваться версией 1.0 приложения, а ваша свежайшая и крутейшая версия 2.0 будет использовать example.com/api/v2

Когда вы делаете обновления, вы практически всегда вносите изменения во внутренние структуры данных и в модели. Это включает изменения в базе данных (добавление или удаление столбцов). Для лучшего понимания давайте представим, что ваш "новый Фейсбук" имеет вызов API, называемый /feed который возвращает объект "Лента". На сегодня, в версии 1, ваш объект "Лента" включает URL аватарки пользователя (avatarURL), имя пользователя (personName), текст записи (feedEntryText) и время создания (timeStamp) записи. Позднее, в версии 2, вы представляете новую возможность, позволяющую рекламодателям размещать описания своих продуктов в ленте. Теперь объект "Лента" содержит, скажем так, новое поле "sourceName", которое перекрывает собой имя пользователя при отображении ленты. Таким образом приложение должно отображать "sourceName" вместо "personName". Поскольку приложению больше не нужно отображать "personName" если задана "sourceName", вы решаете не отправлять "personName" если есть "sourceName". Это все выглядит неплохо до тех пор, пока старая версия вашего приложения, версия 1 не обратится к обновленному серверу. Она будет отображать ваши рекламные записи из ленты без заголовка поскольку "personName" отсутствует. "Грамотный" способ решения такой проблемы — отправлять как "personName", так и "sourceName". Но, друзья, жизнь не всегда так проста. Как разработчик вы не сможете отслеживать все одиночные изменения которые когда либо были произведены с каждой моделью данных в вашем объекте. Это не очень эффективный способ внесения изменений поскольку через пол года вы практически забудете почему и как что-то было добавлено к вашему коду.

Возвращаясь к web 2.0, это не было проблемой вообще. JavaScript клиент немедленно модифицировался для поддержки изменений в API. Однако установленные iOS приложения от вас больше не зависят. Теперь их обновление — прерогатива пользователя.

У меня есть элегантное решение для хитрых ситуаций подобного толка.

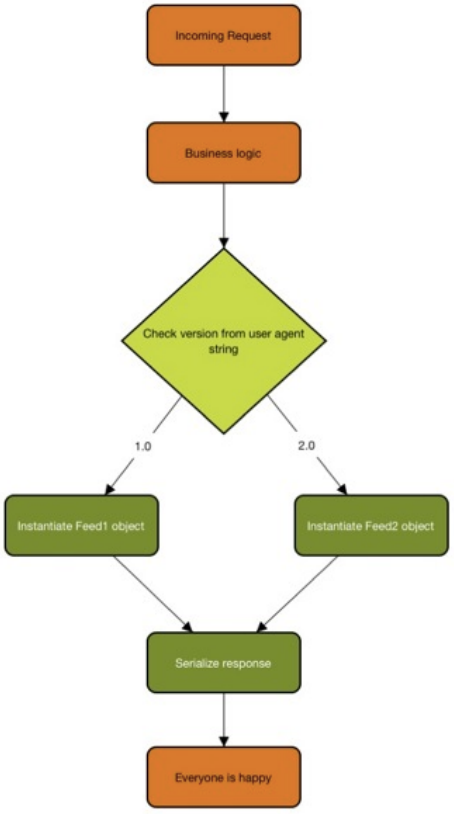
Парадигма разделения на версии через URL

Первое решение — это разделение с использованием URL.
api.example.com/v1/feeds будет использоваться версией 1 iOS приложения тогда как
api.example.com/v2/feeds будет использоваться версией 2.
Несмотря на то, что звучит это все неплохо, вы не сможете продолжать создание копий вашего серверного кода для каждого изменения в формате возвращаемых данных. Я рекомендую использование такого подхода только в случае глобальных изменений в API.

Парадигма разделения на версии через модель

Выше я показал как документировать ваши структуры данных (модели). Рассматривайте эту документацию как контракт между разработчиками серверной и клиентской частей. Вам не следует вносить изменения в модели без изменения версии. Это значит, что в предыдущем случае должно быть две модели, Feed1 и Feed2.

В Feed2 есть поле sourceName и она возвращает sourceName вместо personName если sourceName установлен. Поведение Feed1 остается таким же, как это было оговорено в документации. Алгоритм работы контроллера будет примерно таким:



Вам следует переместить логику создания экземпляра класса в отдельный класс согласно паттерну Factory method. Соответствующий код контроллера должен выглядеть примерно так:

```
Feed myFeedObject = Feed.createFeedObject("1.0");
myFeedObject.populateWithDBObject(FeedDao* feedDaoObject);
```

Где решение о версии используемого API будет принимать контроллер в соответствии с полем UserAgent текста запроса.

Дополнение:
Вместо использования номера версии из строки UserAgent, правильней будет использовать номер версии в заголовке Accept. Таким образом вместо отправки

```
Accept: application/json
```

следует отправлять

```
Accept: application/myservice.1.0+json
```

Таким образом у вас появляется возможность указывать версию API для каждого запроса к REST серверу. Спасибо читателям hacker news за этот совет.

Контроллер использует метод Feed factory для создания корректного объекта feed (лента) основываясь на информации из запроса клиента (все запросы имеют в своем составе поле UserAgent которое выглядит наподобие AppName/1.0) касающейся версии. Когда вы разрабатываете сервер таким образом, любое изменение будет простым. Внесение изменений не будет нарушать имеющиеся соглашения. Просто создавайте новые структуры данных (модели), вносите изменения в factory method для создания экземпляра новой модели для новой версии и все!

При использовании такого подхода ваши приложения версий 1 и 2 могут продолжать работать с одним сервером. Ваш контроллер может создавать объекты версии 1 для старых клиентских приложений и объекты версии 2 для новых.

Прекращение поддержки

С предложенной выше парадигмой разделения API на версии через модель прекращение поддержки вашего API становится намного проще. Это очень важно на последних стадиях когда вы публикуете ваш API. Когда вы делаете глобальное обновление API проведите ревизию всех factory method в ваших моделях в соответствии с изменениями вашей бизнес логики.

Если, в ходе релиза версии 3 вашего API, вы решаете прекратить поддержку версии 1 то для этого достаточно удалить соответствующие модели и удалить строки, создающие их экземпляры в ваших factory method-ах. Создание новых версий и прекращение поддержки старых обязательно будут сопровождать ваш проект показывая насколько он гибок для поддержки ключевых решений, диктуемых бизнесом. Бизнес, неспособный к резким переменам и поворотам, обречен. Обычно неспособность к ключевым переменам обусловлена техническим несовершенством проекта. Указанная техника способна решить такую проблему.

Кэширование

Еще один немаловажный момент, касающийся производительности, которому следует уделить внимание — это кэширование. Если вы считаете, что это задача клиентского приложения подумайте хорошенько. В части 2 этой статьи я расскажу как организовать кэширование, используя средства http 1.1.

Обработка ошибок и интернационализация вашего API

Доведение до клиента причины ошибки в случае ее появления не менее важно чем отправка правильных данных. Я расскажу об обработке ошибок и интернационализации в части 3 данной статьи. Не буду ничего обещать, в любом случае на написание потребуется время.

От переводчика:
Сам я не являюсь разработчиком под iOS и web-сервисов не разрабатывал, мой уровень в этой области можно описать как «Собираюсь стать начинающим». Но тема мне интересна и статья понравилась, причем настолько, что решил перевести.

Вторая часть

restful, web-services

+67 106463 930 Mugunth Kumar ischerbin 0,0

Комментарии (57)

nkuznetsov 17 мая 2012 в 12:17 # +4

Спасибо за статью! Очень актуально для меня именно в данный момент времени. Жду вашего перевода второй части :)

kliss 17 мая 2012 в 12:47 # –4

спеллчекеры уже изобрели.

gnomeby 17 мая 2012 в 14:02 # 0

>> Это в свою очередь значит, что ваши запросы могут кэшироваться любым промежуточным прокси (снижение нагрузки).

Тут надо отметить, что кеш бывает public и private. Поэтому вполне допустимо иметь метод /me/products который для каждого пользователя возвращает разные данные.

ischerbin 17 мая 2012 в 14:39 # ↑ 0

Может и так, но я не автор, я просто перевел в силу возможностей и знаний.

dizer 17 мая 2012 в 19:26 # ↑ 0

Вот кстати у меня вопрос про /me. Как он соотносится со stateless? Ведь равно создается сессия и сервер обрабатывает запрос в контексте сессии пользователя.

lasc 18 мая 2012 в 06:33 # ↑ 0

Какждый раз в запросе передается аунтификация.

rednaxi 17 мая 2012 в 21:25 # ↑ 0

/me/products не соответствует принципу stateless т.к. Подразумевает, что сервер знает что это за me.

Правильно будет /users/gnomeby/products

vladar 18 мая 2012 в 09:51 # ↑ +1

Stateless-принцип просто означает, что сервер не должен хранить и использовать состояние клиента.

«Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server.» — первоисточник

Поэтому если пользователь однозначно идентифицируется через один из заголовков запроса (Autorization, например), то вполне можно использовать /me. Это часть взаимодействия, а не состояние клиента.

В ответ, можно например, сделать редирект на /users/gnomeby/products либо даже напрямую выдавать ответ. В последнем случае нужно не забывать дописать используемый заголовок-идентификатор в Vary ответа.

Clevik 17 мая 2012 в 14:04 # 0

> Тут ссылка на статью по REST на wiki.

Где? :)

ischerbin 17 мая 2012 в 14:35 # ↑ 0

Пропустил, это я себе пометку сделал в процессе. Исправил, спасибо.

seregagl 17 мая 2012 в 14:16 # +1

Переводите вторую часть обязательно

degratnik 17 мая 2012 в 14:42 # −2

А где ссылка на оригинал?

kurumpa 17 мая 2012 в 14:50 # ↑ +7

И так в каждом топике-переводе -_-

ischerbin 17 мая 2012 в 14:51 # ↑ +3

Внизу статьи, под именем автора (не переводчика).

int03e 17 мая 2012 в 18:59 # ↑ 0

На хабре с 2010 года, и до сих пор непонятно где ссылки на оригинал? Вы заебали, товарищи.

Desher 17 мая 2012 в 20:50 # ↑ +3

Это скорее повод сделать ссылку более заметной для читателей

ivakin 17 мая 2012 в 15:15 # 0

спасибо большое за статью, сам тоже начинающий, жду вторую часть…

OneManStartup 17 мая 2012 в 15:25 # +1

Rest можно уже начинать забывать :) hypermedia api скоро будет в тренде

OneManStartup 17 мая 2012 в 15:27 # ↑ 0

Имел ввиду, REST в том смысле в котором его сейчас используют.

cloud 17 мая 2012 в 15:48 # +1

В части передачи пароля какая-то странная каша.
То что вы передаете хэш пароля от клиента который сравнивается с хэшом пароля на сервере, по большому счету тоже самое что передавать пароль текстом и сравнивать его с паролем.
До какой-то степени это может обезопасить пользователей, у которые используют один пароль на всех сайтах, но конкретно вашему сервису это не дает ровным счетом ничего.

dizer 17 мая 2012 в 19:31 # ↑ 0

Дает, если, например, через API нельзя менять пароль.

Злоумышленник получив хэш сможет лишь использовать API, а хозяин, заметив неладное, может зайти через сайт и сменить пароль.

Sharomank 17 мая 2012 в 21:13 # ↑ 0

Вообще хранить сами пароли на сервере это зло, я за хеш, при том что к паролю еще добавляется соль.

Я лично как пользователь перестану пользоваться ресурсом, если узнаю что там хранится мой пароль без изменений.
Да и краснеть не придется, если вашу БД взломают и стянут все логины с паролями.
Ну и самое главное нужно думать о пользователях, а большинство такое, что логины и пароли у них повторяются.

petrkozorezov 17 мая 2012 в 20:04 # +1

Статья хорошая, продолжение было бы очень кстати.

spmbt 17 мая 2012 в 20:08 # 0

То, что касается разделения приложений на версии, я подробно расписывал в habrahabr.ru/post/133989/ («Система настроек и смена версий программ»). Тогда, видимо, было воспринято как очень сложное, здесь будет ближе к теме.

TimReset 17 мая 2012 в 21:41 # 0

Понимаю, что перевод, поэтому вопрос не автору, а переводчику и знающим людям :-)
В статье сказано:
>DELETE абсолютно однозначен. Он индемпотентен как и PUT, и должен использоваться для удаления записи если таковая существует.
Объясните, пожалуйста, почему DELETE считается индемпотентным. Ведь после второго вызова объект не удалится (если при первом вызове удалился) и это будет уже не равнозначные вызовы.

dizer 17 мая 2012 в 23:00 # ↑ +1

DELETE products/last — плохо
DELETE products/542 — хорошо

VolCh 18 мая 2012 в 03:01 # ↑ 0

После двух подряд одинаковых вызовов DELETE состояние хранилища не изменится, как и при двух подряд одинаковых вызовах PUT, в отличии от вызовов POST.

vladar 18 мая 2012 в 09:39 # ↑ 0

Мы часто путаемся из-за того, что есть состояние клиента и состояние сервера.

Собственно, «idempotence» в мире REST чаще всего используется по отношению к состоянию сервера и означает, что при нескольких применениях одного и того же метода сервер остается в одном и том же состоянии. Хотя по факту на клиенте вы можете получить разные ответы.

А тот же stateless-принцип просто означает, что нельзя держать состояние клиента на сервере (а точнее — менять поведение сервера в зависимости от состояния клиента).

savostin 17 мая 2012 в 23:18 # +1

Про версионность моделей красиво.
А что делать с версионностью базы данных?
Как поддерживать старую версию, если структуру надо менять?

VolCh 18 мая 2012 в 03:06 # ↑ 0

Маппинг менять, если изменения вроде переноса столбца в другую таблицу. Хуже когда, например, нужно преобразование связи один-к-одному в один-ко-многим, тут уже волевое решение надо применять о том, что будет отдаваться в старой версии.

mgrach 18 мая 2012 в 07:31 # ↑ 0

Можно использовать документо-ориентированные БД, они ж scheme-free. Это поможет, если структуру нужно только дополнять.

vladar 18 мая 2012 в 10:14 # ↑ 0

Красивая версионность — это в HTML. Будущее версионности API видимо тоже за подобным подходом — когда модель будет дополнительно описываться некоторыми тэгами или другими метаданными. При этом незнакомые тэги будут игнорироваться клиентом, а в случае несовместимых изменений будут вводиться новые тэги. Тогда старые клиенты смогут продолжать работу хотя бы частично.

kushti 18 мая 2012 в 00:08 # 0

Интересно, а какое отношение к технологии Rest имеет этот хипстерский пафос,- «В 2007-м Стив Джобс представил iPhone, который произвел революцию в высокотехнологичной индустрии и изменил наш подход к работе и ведению бизнеса. »?

Zelgadis 18 мая 2012 в 07:04 # ↑ 0

Тем, что RESTful API нужен для приложений, и все хипстеры-стартаперы обязаны иметь rest api для свего приложение и ios приложение которое использует это api, иначе в инстаграме забанят.

milast 18 мая 2012 в 02:37 # –1

«Что как бы характеризует отношение автора к PHP.» Ваше отношение к PHP мало кого интересует. Мне нравится PHP, Java, но НЕ нравится Ruby, но я никогда не отзывался о последнем отрицательно, потому как это сугубо личное дело каждого. Достаточно было просто опустить этот момент.

В целом статья интересна. Спасибо.

ischerbin 18 мая 2012 в 05:44 # ↑ 0

Несмотря на *«Ваше отношение к PHP мало кого интересует»* я все таки скажу, что сам пишу на PHP и вполне им доволен, ничего против не имею и холиварить не собирался. Это небольшая попытка юмора автора, которую я в свою очередь попытался перевести.
Обновил статью, добавив ссылку на тот пост, там досталось всем языкам, не только PHP. Так что не сочтите мое отступление оскорблением

VolCh 18 мая 2012 в 03:09 # 0

С аутентификацией конечно в REST сложно. Столько статей прочёл, но так и не понял как правильно её делать.

mgrach 18 мая 2012 в 07:34 # ↑ 0

Тоже этот вопрос волнует. Тем более, что у меня в разработке сервис, в котором данные наполняются машинами, а читаются людьми. И тех и тех нужно авторизовывать.

vladar 18 мая 2012 в 10:06 # ↑ 0

А в чем проблема? Объясните свои затруднения. Авторизация не есть «состояние клиента» — это обязательная часть взаимодействия.

Поэтому OAuth, HTTP Basic, HTTP Digest — всё это правильные способы аутентификации для REST. Даже подписанная Cookie, содержащая ID пользователя подойдет.

С Cookie другая проблема — она часто используется именно для передачи ID сессии, хранящей клиентское состояние. Но если вы не используете сессию, то Cookie можно использовать как «workaround» против невозможности скормить браузеру собственный заголовок Authorization.

Если у вашего API это задокументировано, то всё нормально. Из той же области, что и использование параметра _method с POST запросом, когда PUT и DELETE не поддерживаются браузером.

VolCh 18 мая 2012 в 13:40 # ↑ 0

Я скорее не про выделенное API для клиентов, а про обычный сайт, использующий принципы REST, где API получается как следствие простым использованием GET /posts/%id%.json вместо GET /posts/%id%.html. И да, неразрывно связывая аутентификацию и сессию.

Yan169 18 мая 2012 в 11:17 # +1

Я не против REST подхода в целом, но нужно понимать, что он имеет и свои недостатки, в том числе и для примера в статье:

> Что может пойти не так в приведенном случае, так это то, что если ваш пользователь использует сервис с двух или трех устройств, то, в случае когда одно из них устанавливает последний прочитанный элемент, то остальные не смогут загрузить элементы ленты, прочитанные на других устройствах ранее.
>
> Независимость от состояния означает, что данные, возвращаемые определенным вызовом API, не должны зависеть от вызовов, сделанных ранее.

Зато при REST пользователь прочтя ленту на одном клиенте, при переходе на другой должен заново просматривать всю ленту в поисках последнего прочитанного элемента, что может занять чуть ли не больше времени, чем собственно просмотр новых элементов. Пример — твиттер.

danSamara 20 мая 2012 в 23:32 # ↑ 0

В статье написано верно — клиент должен получать сообщения относительно временной метки, дабы загружать _все_ твиты. Иначе, при запросе, допустим /get-last, он будет получать только новые и в ленте будут пробелы.
Другое дело, что каждое сообщение должно помечаться меткой прочитано/не_прочитано, тогда загрузятся все сообщения, а непрочитанные будут выделены в интерфейсе клиента.

Yan169 21 мая 2012 в 00:14 # ↑ 0

> Другое дело, что каждое сообщение должно помечаться меткой прочитано/не_прочитано, тогда загрузятся все сообщения, а непрочитанные будут выделены в интерфейсе клиента.

В таком варианте опять-таки GET-запрос /get-last будет во-первых изменять состояние сервера, а во вторых ответ на этот запрос будет зависеть от предыдущих вызовов. И то и то противоречит REST.

danSamara 21 мая 2012 в 00:27 # ↑ 0

Я наверное некорректно выразился: /get-last — неправильный запрос. Надо использовать описанный в статье /feed?lastFeed=20120228, то есть с указанием диапазона получаемых сообщений.
В этом случае состояние сервера не изменяется.

Yan169 21 мая 2012 в 01:00 # ↑ 0

Не важно, какой запрос. Если состояние сервера от запроса не изменится, то сервер не запомнит, какие сообщения были прочитаны, и не сможет их пометить при следующем запросе (но с другого устройства).

danSamara 21 мая 2012 в 01:02 # ↑ 0

Всё верно. Пока пользователь сообщение не прочитал — оно не прочитано. Когда прочитал — делаем PUT на сервер и помечаем как прочитанное. Что не так?

Yan169 21 мая 2012 в 01:18 # ↑ 0

Это и есть недостаток. Дополнительный запрос на сервер там, где без него можно элементарно обойтись.
И даже в этом случае всё равно формально принцип statelessness соблюдаться не будет: на один и тот же запрос /feed?lastFeed=20120228 могут приходить разные ответы (в одном одни сообщения помечены как непрочитанные, в другом — другие).

danSamara 21 мая 2012 в 01:51 # ↑ 0

Давайте лучше на примере, а то мы, возможно, говорим немного о разном.

Пусть будет приложение для чтения RSS ленты (типа google reader). Есть декстоп и мобильное приложение для планшета. Есть лента с пронумерованными постами (нумерация увеличивается по мере добавления сообщений).

В ленте 100 сообщений, 20 — непрочитанных.
На десктопе отображается: 100 сообщений (20 новых) — актуальная версия.
На планшете: 70 сообщений (0 новых) — до обновления данных.

Обновляю данные на планшете: GET /posts?last=70, получаю: {all:30, new:20}.
Если я выполню запрос GET /posts?last=70 ещё раз, ответ будет точно таким же, т.к. данные не изменились.
Читаю одно сообщение: PUT /post/81?read=1
Планшет обновляется: GET /posts?last=81, ответ: {all:19, new: 19}

Переключаюсь на десктоп, он это палит и запрашивает обновление: GET /posts/last=80, получает: {all:20, new: 19}

Из примера видно:
— состояние сервера не меняется, клиенты сами знают сколько у них сообщений сейчас у сами формируют запрос на получение актуальной информации.
— на один и тот же запрос будет приходить один и тот же ответ до тех пор, пока не изменяться данные в БД

Вопросы к вам:
— почему вы считаете, что ответы разные на один и тот же запрос?
— как можно обойтись без PUT (вы ведь его имели ввиду под словами «лишний запрос»)?

Yan169 21 мая 2012 в 02:40 # ↑ 0

У вас в примере всё хорошо (почти). Но я твиттер как пример приводил, для которого просто глупо на каждый твит слать отдельный запрос.
Все твиты логичнее слать в одном запросе, при REST это:
`планшет GET /posts?last=70, ответ: {{tweet71},{tweet72},{tweet100}}`
Потом, пока я переключаюсь на десктоп, приходит ещё десяток-другой твитов, кроме того, последний раз на десктопе я с утра читал только 10й твит. Получаем
`десктоп GET /posts?last=10 ответ:{{tweet11},...,{tweet120}}`
Итого, вместо последних 20 твитов, я получаю 110, среди которых я вынужден искать последний прочитанный. И поиск зачастую длится дольше просмотра непрочитанных элементов. Пример, кстати, не синтетический, а отражает личный опыт чтения твиттера на iPad и десктоп.
Можно конечно и PUT-запрос слать:
`планшет PUT /posts?lastread=100`
Но зачем? Ради формального соответствия REST?

Кроме того, при таком API я всё равно получу 110 элементов, пусть они и будут разделены на прочитанные/новые.
А в идеале мне, как пользователю, нужен вариант:
`планшет GET /posts?new, ответ: {{tweet70,read:1},{tweet71:read:0},{tweet72,read:0},{tweet100,read:0}}`
`десктоп GET /posts?new ответ:{{tweet100,read:1},...,{tweet120,read:0}}`
Т.е. грузятся только новые плюс один-два-три старых твита, чтоб вспомнить, на чем остановился читать. Строгий REST этого обеспечить не может.

enej 21 мая 2012 в 06:38 # ↑ 0

Вы предлагает помечать сообщение как прочитанные при их загрузке? То есть если телефон автоматически синхронизирует ленту, но я ее не стану смотреть, то на десктопе, я этих твитов уже не увижу?

Yan169 21 мая 2012 в 10:16 # ↑ 0

Да, я предлагаю помечать сообщения как прочитанные в момент, когда они показываются пользователю в открытом виде, т.е. если я набираю twitter.com, подгружаются твиты, то они должны быть помечены как прочитанные.

> То есть если телефон автоматически синхронизирует ленту, но я ее не стану смотреть, то на десктопе, я этих твитов уже не увижу?

Что значит «не увижу»? Пролистнули ленту вниз, и подгрузились более ранние сообщения. Никто же не предлагает ограничивать API одним `/posts?new`
Вы рассматриваете клиент как архиватор абсолютно *всех* сообщений? Я — как средство удобного слежения за непрочитанными. Для первого REST подходит полностью, для второго — нет.

enej 21 мая 2012 в 13:11 # ↑ 0

Так чтобы подгрузились ранние сообщения, нужно будет делать еще один запрос и в чем тогда преимущество вашего решения? Кроме, того пользователю будут показываться не с реально последнего прочитанного сообщения, а с последнего загруженного и все равно прийдется листать, только теперь не вперед, а назад. Пользователями API могут быть не только люди, но и другие программы, а в таком случае может и не быть потребности отмечать загруженные сообщения, как прочитанные.

Yan169 21 мая 2012 в 13:29 # ↑ 0

Преимущество в том, что ранние (вероятнее всего прочитанные) сообщения подгружать нужно редко. См. пример 20 твитов vs 110 твитов.
В целом предлагаю проектировать API ориентируясь в первую очередь на бизнес-требования (чтобы пользователям было удобно), а не на удобство программисту и следование формальным стандартам. Точнее так: следовать стандарту, но осознавать все его недостатки, и в случае необходимости делать исключения.

> может и не быть потребности

`/posts?last=70&markasread=0` например решит проблему

enej 21 мая 2012 в 14:24 # ↑ 0

«вероятнее всего прочитанные» — у меня на телефоне автоматическая синхронизация включена, на десктопе тоже запущен клиент, который постоянно тянет новые твиты. Так, что в моем случае вероятнее всего новые сообщения будут разбросаны между устройствами.
API он для программистов, а не для пользователей.
Но с выводом, что не нужно слепо следовать стандарту, согласен.

eugzol 15 января 2015 в 13:39 # ↑ 0

> В таком варианте опять-таки GET-запрос /get-last будет во-первых изменять состояние сервера

Вот это философское «изменение состояние сервера» отличный аргумент :) Надо попросить Гугл начать соответствовать этой концепции в своих API, и перестать, например, начислять квоту на запросы:

developers.google.com/youtube/v3/docs/playlists/list?hl=ru#part

Разработчик должен сам вдогонку к каждому подобному GET-запросу отправлять POST на уменьшение квоты, производя расчёты по приведённым расценкам! :)

Flcn 20 мая 2012 в 00:04 # 0

Я надеюсь про HATEOAS во второй части будет инфрмация?

ischerbin 21 мая 2012 в 10:26 # ↑ 0

Будет, но кратко в начале. Планирую до конца недели выложить вторую часть. Перевод готов, осталось оформить.

Только зарегистрированные пользователи могут оставлять комментарии. Войдите, пожалуйста.