Лучшие практики проектирования REST API

Определение

Для начала нужно определиться, что же такое REST. Википедия даёт на этот вопрос следующий ответ. **REST** (**Representational State Transfer** — «передача состояния представления») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы.

Своими словами я бы объяснил понятие REST как “набор рекомендаций, который позволяет унифицировать взаимодействие клиентских и серверных приложений”.
В данной статье я постараюсь рассказать об этих самых “рекомендациях”, которые помогут проектировать и создавать REST-сервисы согласно общепринятым практикам.

Также нужно понимать, что такое REST-сервис. Я бы дал определение REST-сервису, как “точка взаимодействия клиентского приложения с сервером”. Говоря Java терминологией - это сервлет, на который клиент посылает запрос.

Проблематика

Но прежде чем начинать описывать правила, я хотел бы донести мысль о том, что REST - это не стандарт, потому нет единых строгих правил, которых стоит придерживаться. Это значит, что до сих пор нет полной согласованности о том, какие решения лучше применять в той или иной ситуации. Очень часто заходят споры о том, какие HTTP методы использовать и какой HTTP код возвращать в каждой конкретной ситуации.

Более подробно о проблемах REST можно почитать тут.
Соответственно, рекомендации, которые я опишу ниже, стоит принимать во внимание, но не следовать им на 100%, если это может повредить какому-либо другому аспекту проекта.

Однако если вы будете следовать всем рекомендациям, то получится спроектировать очень понятную и поддерживаемую систему, с которой приятно работать.

Название сервиса

Для начала необходимо выбрать имя для REST сервиса. Под именем сервиса я подразумеваю его путь в URI запросе. Например, http://my-site.by/api/rest/service/name. Для выбора имени нам нужно понимать что такое “ресурсы” в архитектуре REST.

Представление ресурса

В терминологии REST что угодно может быть ресурсом — HTML-документ, изображение, информация о конкретном пользователе и т.д. Если ресурс представляет собой некоторый объект, его легко представить, используя некоторый стандартный формат, например, XML или JSON. Далее сервер может отправить данный ресурс, используя выбранный формат, а клиент сможет работать с полученным от сервера ресурсом, используя этот же формат.

Пример представления ресурса “профиль” в формате JSON:

```
{
    "id":1,
    "name":"Mahesh",
    "login":"manesh"
}
```

REST не накладывает явных ограничений на формат, который должен быть использован для представления ресурсов, но есть ряд правил, которым нужно следовать при разработке формата, который будет использоваться для представления ресурса:

- Клиент и сервер должны “понимать” и иметь возможность работать с выбранным форматом.
- Ресурс можно полностью описать, используя выбранный формат независимо от сложности ресурса.
- Формат должен предусматривать возможность представления связей между ресурсами.

Пример представления ресурса “заказ” и его связи с ресурсом “профиль”:

```
{
    id: 11254,
    currency: "EUR",
    amount: 100,
    profile: {
        id: 11,
        uri: "http://MyService/Profiles/11"
    }
}
```

Как видно, не обязательно полностью дублировать всю структуру ресурса, на который ссылается другой ресурс. Вместо этого можно использовать понятную ссылку на другой ресурс.

Обращение к ресурсу

Каждый ресурс должен быть уникально обозначен постоянным идентификатором. «Постоянный» означает, что идентификатор не изменится за время обмена данными, и даже когда изменится состояние ресурса. Если ресурсу присваивается другой идентификатор, сервер должен сообщить клиенту, что запрос был неудачным и дать ссылку на новый адрес. Каждый ресурс однозначно определяется URL. Это значит, что URL по сути является первичным ключом для единицы данных. То есть, например, вторая книга с книжной полки будет иметь вид ***/books/2***, а 41 страница в этой книге — ***/books/2/pages/41***. Отсюда и получается строго заданный формат. Причем совершенно не имеет значения, в каком формате находятся данные по адресу ***/books/2/pages/41*** – это может быть и HTML, и отсканированная копия в виде jpeg-файла, и документ Word.

Рекомендуется при определении имени REST-сервиса использовать имена ресурсов во множественном числе. Такой подход позволяет добавлять новые REST-сервисы лишь расширяя имена уже существующих. Например, сервис ***/books*** вернёт нам список всех книг, ***/books/3*** вернёт информацию о 3-ей книге, а сервис ***/books/3/pages*** вернёт все страницы 3-ей книги.

Для сервисов, которые выполняют какие-то специфические действия над ресурсом, есть 2 подхода для указания действия: в имени сервиса или в его параметрах. Например, ***/books/3/clean или /books/3?clean***. Я предпочитаю первый вариант, так как обычно такие сервисы не редко используют POST методы, которые не поддерживают передачу параметров в URI, что делает сервис, на мой взгляд, не очень читабельным. Используя определение типа действия в имени сервиса, мы делаем наш сервис более расширяемым, так как он не зависит от типа HTTP метода.

Также очень не рекомендуется использовать имена, включающие в себя несколько слов и описывающие бизнес составляющую сервиса (как это рекомендуется делать при именовании java методов). Например, вместо ***/getAllCars*** лучше сделать метод ***/cars***. Если же метод нельзя никак описать одним словом, то необходимо применять единый стиль разделителей, я обычно использую ‘-’, что является наиболее популярным подходом. Например, ***/cars/3/can-sold.***

Более подробно о проектировании названий REST-сервисов можно прочитать в этой статье.

HTTP методы

Далее нам необходимо выбрать HTTP метод, который будет использовать наш REST-сервис, ведь имея даже одинаковое имя, но разные методы, REST-сервисы выполняют совершенно различные действия.

В REST используются 4 основных HTTP метода: GET, POST, PUT, DELETE. В большинстве случаев каждый из методов служит для выполнения предопределённого ему действия из CRUD (***c****reate,* ***r****ead,* ***u****pdate,* ***d****elete — «создание, чтение, обновление, удаление»*).
POST - create, GET - read, PUT - update, DELETE - delete.

ВАЖНОЕ ДОПОЛНЕНИЕ: *Существуют так называемые REST-Patterns, которые различаются связыванием HTTP-методов с тем, что они делают. В частности, разные паттерны по-разному рассматривают POST и PUT. Однако, PUT предназначен для создания, замены или*

обновления, для POST это не определено (The POST operation is very generic and no specific meaning can be attached to it). Поэтому иногда POST и PUT можно поменять местами. Но в большинстве случаев POST используют для создания, а PUT для редактирования, и чуть позже я объясню почему.

Приведу несколько примеров использования различных методов для взаимодействия с ресурсами.

- *GET /books/* – получает список всех книг. Как правило, это упрощенный список, т.е. содержащий только поля идентификатора и названия объекта, без остальных данных.
- *GET /books/{id}* – получает полную информацию о книге.
- *POST /books/* – создает новую книгу. Данные передаются в теле запроса.
  PUT /books/{id} – изменяет данные о книге с идентификатором {id}, возможно заменяет их. Данные также передаются в теле запроса.
- *OPTIONS /books* – получает список поддерживаемых операций для указанного ресурса (практически не используется)
- *DELETE /books/{id}*– удаляет данные с идентификатором {id}.

Безопасность и идемпотентность

Очень помогут в выборе HTTP метода знания о безопасности и идемпотентности этих методов.

Безопасный запрос — это запрос, который не меняет состояние приложения.

Идемпотентный запрос — это запрос, эффект которого от многократного выполнения равен эффекту от однократного выполнения.

Http-метод	Безопасный	Идемпотентный
GET	Да	Да
PUT	Нет	Да
DELETE	Нет	Да
POST	Нет	Нет
OPTIONS	Да	Да

Судя по данной таблице, GET-запрос не должен менять состояние ресурса, к которому применяется. PUT и DELETE запросы могут менять состояние ресурса, но их можно спокойно повторять, если нет уверенности, что предыдущий запрос выполнился. В принципе, это логично: если многократно повторять запрос удаления или замены определенного ресурса, то результатом будет удаление или замена ресурса. Но POST запрос, как мы видим из таблицы, небезопасный и неидемпотентный. То есть мало того, что он меняет состояние ресурса, так и многократное его повторение будет производить эффект, зависимый от количества повторений. Ему по смыслу

соответствует операция добавления новых элементов в БД: выполнили запрос X раз, и в БД добавилось X элементов.

Также приведу пример того, почему GET-запросы не должны изменять состояние ресурса. GET-запросы могут кэшироваться, например, на уровне прокси-сервера. В таком случае запрос может даже не дойти до сервера приложения, а в качестве ответа прокси-сервер вернёт информацию из кэша.

HTTP коды

В стандарте HTTP описано более 70 статус кодов. Хорошим тоном является использование хотя бы основных.

- 200 – OK – успешный запрос. Если клиентом были запрошены какие-либо данные, то они находятся в заголовке и/или теле сообщения.
- 201 – OK – в результате успешного выполнения запроса был создан новый ресурс.
- 204 – OK – ресурс успешно удалён.
- 304 – Not Modified – клиент может использовать данные из кэша.
- 400 – Bad Request – запрос невалидный или не может быть обработан.
- 401 – Unauthorized – запрос требует аутентификации пользователя.
- 403 – Forbidden – сервер понял запрос, но отказывается его обработать или доступ запрещен.
- 404 – Not found – ресурс не найден.
- 500 – Internal Server Error – разработчики API должны стараться избежать таких ошибок.

Эти ошибки должны быть отловлены в глобальном catch-блоке, залогированы, но они не должны быть возвращены в ответе.

Чем обширнее набор кодов, который мы будем использовать, тем более понятный будет API, который мы создаём. Однако нужно учесть, что некоторые коды браузеры обрабатывают по-разному. Например, некоторые браузеры получив код ответа 307 сразу же выполняют редирект, а некоторые позволяют обработать такую ситуацию и отменить действие. *Прежде чем использовать тот или иной код, необходимо полностью понимать, как он будет обрабатываться на клиентской стороне!*

Более подробно о HTTP кодах можно почитать тут.

Headers

Рекомендуется при проектировании REST-сервисов явно указывать заголовки, в которых обозначен формат обмена данными:

- *Content-Type* - формат запроса;
- *Accept* - список форматов ответа.

Параметры поиска ресурсов

Чтобы упростить использование сервисов, отвечающих за возвращение какой-либо информации, и вдобавок сделать их наиболее производительными, необходимо использовать в качестве параметров запроса параметры для сортировки, фильтрации, выбора полей и пагинации.

Фильтрация

Используйте уникальный параметр запроса для каждого поля, чтобы реализовать фильтрацию. Это позволит ограничить количество выводимой информации, что оптимизирует время обработки запроса.

Например, чтобы вывести все красные книги необходимо выполнить запрос:

GET /books?color=red

Сортировка

Сортировка реализуется подобно фильтрации. Например, чтобы вывести все книги, отсортированные по году публикации по убыванию и по названию по возрастанию нужно выполнить следующий запрос:

GET /books?sort=-year,+name

Пагинация

Для того, чтобы поддержать возможность загрузки списка ресурсов, которые должны отображаться на определённой странице приложения, в REST API должен быть предусмотрен функционал пагинации. Реализуется он с помощью знакомых нам по SQL параметрам limit и offset. Например:

GET /books?offset=10&limit=5

Помимо того хорошим тоном является вывод ссылок на предыдущую, следующую, первую и последнюю страницы в хидере Link. Например:

Link: <http://localhost/api/books?offset=15&limit=5>; rel="next",
<http://localhost/api/books?offset=50&limit=3>; rel="last",
<http://localhost/api/books?offset=0&limit=5>; rel="first",
<http://localhost/api/books?offset=5&limit=5>; rel="prev"

Рекомендуется также возвращать общее количество ресурсов в хидере *X-Total-Count.*

Выбор полей ресурса

Для более удобного использования сервиса, для экономии трафика можно предоставить возможность управлять форматом вывода данных. Реализуется предоставлением возможности выбора полей ресурса, которые должен вернуть REST сервис. Например, если необходимо получить только id книг и их цвета, необходимо выполнить следующий запрос:

GET /books?fields=id,color

Хранение состояния

Одно из ограничений RESTful сервисов заключается в том, что они не должны хранить состояние клиента, от которого получают запросы.

Пример сервиса, не хранящего состояние:
Request1: ***GET http://MyService/Persons/1 HTTP/1.1***
Request2: ***GET http://MyService/Persons/2 HTTP/1.1***

Каждый из этих запросов может быть обработан независимо от другого.

Пример сервиса, хранящего состояние:
Request1: ***GET http://MyService/Persons/1 HTTP/1.1***
Request2: ***GET http://MyService/NextPerson HTTP/1.1***

Чтобы обработать второй запрос, серверу потребуется “запомнить” id последнего человека, который был запрошен клиентом. Т.е. сервер должен “запомнить” свое текущее состояние, иначе второй запрос не может быть обработан. При проектировании сервиса, следует избегать необходимости в хранении состояния, так как это имеет ряд преимуществ.

Преимущества сервиса, не хранящего состояние:

- сервис обрабатывает запросы независимо друг от друга;
- архитектура сервиса упрощается;
- не требуется дополнительных усилий для реализации сервисов с использованием протокола HTTP, который также не хранит состояния.

Недостатки сервиса, не хранящего состояние:

- клиент сам должен отвечать за передачу необходимого контекста сервису.

Версионность

Хорошим тоном является поддержка версионности REST API. Это позволит в дальнейшем легко расширять API, без обязательного внесения изменений в клиенты, которые уже пользуются им. Имеются несколько подходов реализации версионности:

- С использованием Accept хидера. В данном случае версия API указывается в Accept - **Accept:text/v2+json**
- С использованием URI. В таком подходе версия API указывается прямо в URI - http://localhost/api/v2/books
- Использование кастомного хидера. Можно использовать собственный хидер, который будет отвечать только за передачу версии API **- API-Version:v2**
- Использование параметра запроса. Можно использовать параметр запроса для передачи версии API - **/books?v=2**

Каждый из представленных способов имеет право на существование, у каждого есть свои плюсы и минусы. Однако только Вам решать, какой способ реализации версионности подойдёт Вашему проекту.

Документация

Для удобного пользования нашими REST сервисами нужно создать хорошую и понятную документацию. Для этих целей можно использовать различные инструменты, например, Mashape или Apiary, но я рекомендую использовать Swagger.

Swagger - это технология, которая позволяет документировать REST-сервисы. Swagger поддерживает множество языков программирования и фреймворков. Плюс, Swagger предоставляет UI для просмотра документации.

Получить более подробную информацию о Swagger можно по данной ссылке.

Архивирование

Для экономии трафика рекомендуется использовать архивирование, при передаче больших данных. Это сократит время выполнения запроса. В большинстве современных фреймворков данная функция реализована по умолчанию.

Кэширование

Также для сокращения запросов к БД и увеличения быстродействия наших REST сервисов рекомендуется применить механизм кэширования. Кэширование можно настраивать как на уровне сервера, так и в самом приложении, в зависимости от ситуации.

Кэшированием можно управлять используя следующие HTTP заголовки:

- Date - дата и время создания ресурса.
- Last Modified - дата и время последнего изменения ресурса на сервере.
- Cache-Control - заголовок HTTP 1.1 используемый для управления кэшированием.
- Age - время, прошедшее с момента последнего получения ресурса, заголовок может быть добавлен промежуточным (между клиентом и сервером) компонентом (например, прокси

сервер)

Рекомендации по кэшированию:

- Рекомендуется кэшировать статические ресурсы, такие как изображения, стили css, файлы javascript.
- Не рекомендуется указывать большое время жизни кэша.
- Динамическое содержимое должно кэшироваться на короткое время или не кэшироваться вообще.

Общие рекомендации

Тип представления ресурса

Хотя REST не накладывает явных ограничений на формат, который должен быть использован для представления ресурсов, наиболее популярными являются XML и JSON.

Существует множество библиотек в разных языках программирования для удобной работы с этими форматами. Несмотря на это, рекомендуется использовать именно JSON для представления ресурсов. Это более легкий, читабельный формат, с которым проще работать по сравнению с XML, а так же проще выполнять сериализацию/десериализацию объектов в различных языках программирования.
Однако иногда для поддержки некоторых REST клиентов сервису необходимо поддерживать XML формат. В таком случае можно реализовать поддержку обоих форматов и указывать в параметре запроса, в каком формате должен быть представлен ответ.

Использование проверенных решений для авторизации

Используйте для авторизации проверенные и отработанные схемы.
Не изобретайте свой велосипед с использованием md5 подписей данных и прочего, доверьтесь профессионалам в области защиты данных. Всё уже придумано за Вас: OAuth, OpenID, APIKeys.

Обработка исключений

При возникновении ошибочных ситуаций необходимо выводить отформатированную и понятную информацию. Это относится в первую очередь к статус коду в HTTP ответе. Ошибки в сервисах чаще всего относятся к двум типам:

- 4xx - ошибки клиента;
- 5xx - ошибки сервера.

В случае ошибки клиента, например, ошибки валидации какого-то из параметров запроса, в теле ответа рекомендуется передавать полезную информацию об ошибке: сообщение, описание, код (например в формате JSON). В случае ошибки сервера отправлять дополнительную информацию в теле ответа не всегда возможно, например, в случаях когда сервер не доступен.

Плохим тоном является вывод всего стэк трейса исключения. Рекомендуется для каждой исключительной ситуации иметь свой код. В дальнейшем при выводе информации об ошибке можно будет добавить ссылку на документацию, в которой данный код будет выступать уникальным идентификатором.
Например:

```
1. {
2.     "code" : 1234,
3.     "message" : "Something bad happened :(",
4.     "description" : "More details about the error here",
5.     “moreInfo”: “http:/localhost/api/v2/errors/1234”
6. }
```