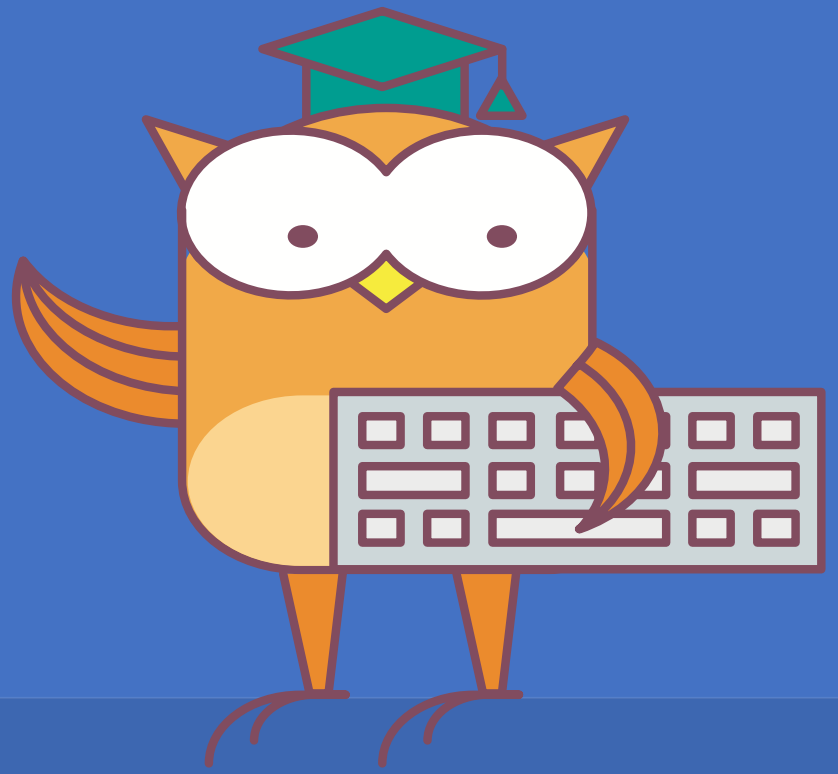
Меня хорошо слышно && видно?

Напишите в чат, если есть проблемы!

Ставьте + если все хорошо

RESTful API паттерны

Архитектор ПО



Щетинников Стас

Карта вебинара

- REST. Maturity Levels
- RESTful Patterns
- JsonScheme, OpenAPI

00

HTTP

Структура HTTP/1.1

- Стартовая строка (разная для запроса и ответа)
- Заголовки + пустая строка
- Тело

```
МЕТОД /some/uri HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 122
…
Название: значение
(пустая строка)
Тело запроса (json, html, текст…)
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 122
…
Название: значение
(пустая строка)
Тело запроса (json, html, текст…)
```

Передача параметров

- Path - `/article/42/comment-from/john`

  `/article/{articleId}/comment-from/{user}`
- Query - `/some/uri/?article=42&comment-from=john`
- Header - `x-article: 42`
- Cookie - `Cookie: articleId=42; commentFrom=john`
- Тело - `{“article”: 42, “commentFrom”: “john”}`

Ограничения

- Path + Query - до 2KB
- Path - числа или строки, человекочитаемый вид
- Header до 8KB (иногда вместе с Path). Тут же jwt токен!
- Тело - отсутствует в некоторых методах

HTTP методы (часто используемые)

- GET - запросы на чтение
- POST - rpc-вызов (вот данные, обработай), иногда запрос на чтение со сложным фильтром
- PUT - изменение или создание новой сущности
- PATCH - изменение только части полей сущности
- DELETE - удаление ресурса

Идемпотентные: GET, PUT, DELETE

HTTP методы (часто используемые)

- GET - запросы на чтение
- POST - rpc-вызов (вот данные, обработай), иногда запрос на чтение со сложным фильтром
- PUT - изменение или создание новой сущности
- PATCH - изменение только части полей сущности
- DELETE - удаление ресурса

Идемпотентные: GET, PUT, DELETE

Статусы HTTP

https://httpstatuses.com/

- 200 - OK
- 207 - “OK, но с предупреждениями”
- 4xx - ошибка клиента
- 5xx - ошибка сервера

01

REST

REST API

Раз HTTP используется в Internet и есть огромное количество библиотек под него и целая инфраструктура, почему не построить взаимодействие между сервисами на основе HTTP, максимально используя его возможности и максимально идиоматично.

Об этом подумал Рой Филиднг один из создателей Интернета, и написал целую диссертацию, как межсервисное взаимодействие переложить на HTTP.

RESTful API

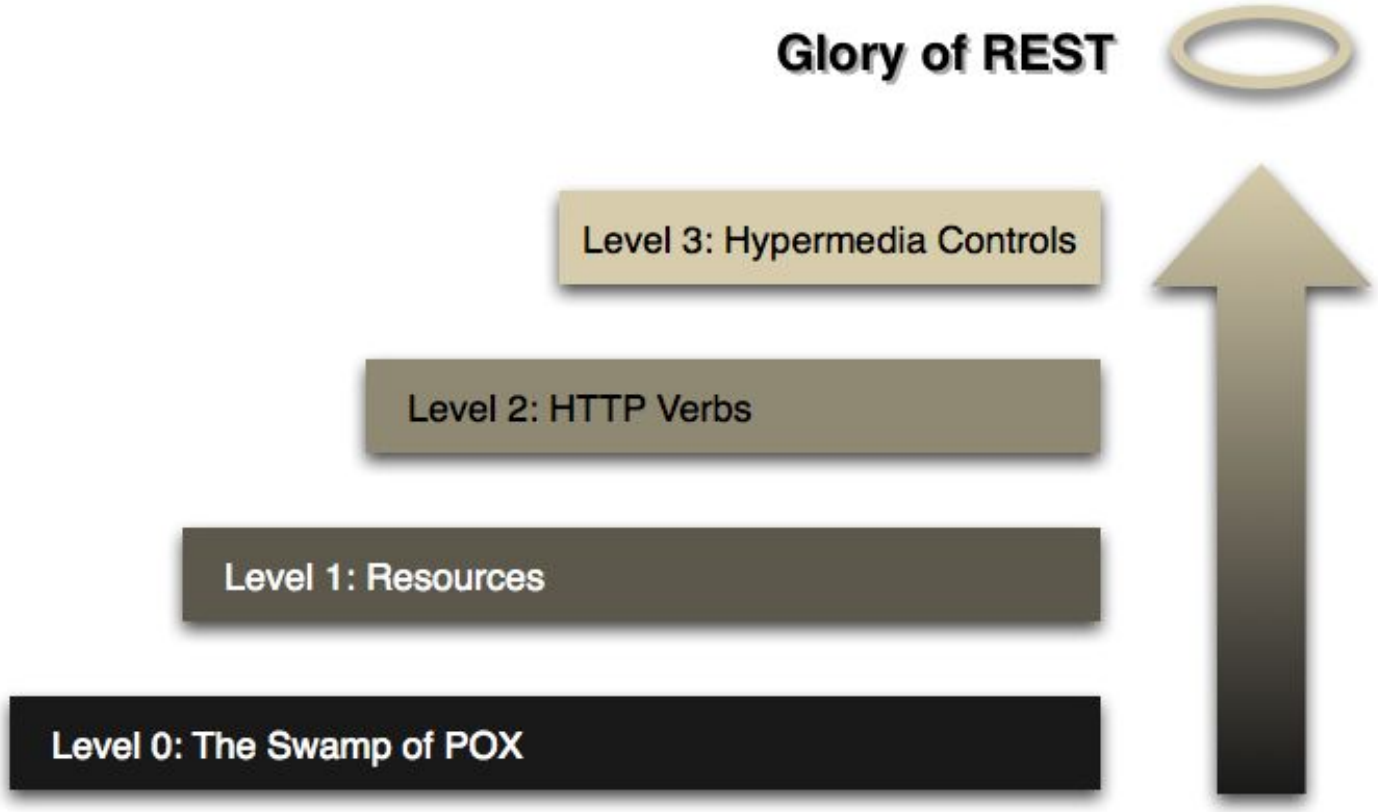
RESTful API – это подход к проектированию API, в котором API - это набор ресурсов, которые являются представлением сущностей предметной области, и HTTP глаголов для манипулирования ими.

REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Уровни зрелости REST

Модель зрелости Ричардсона.

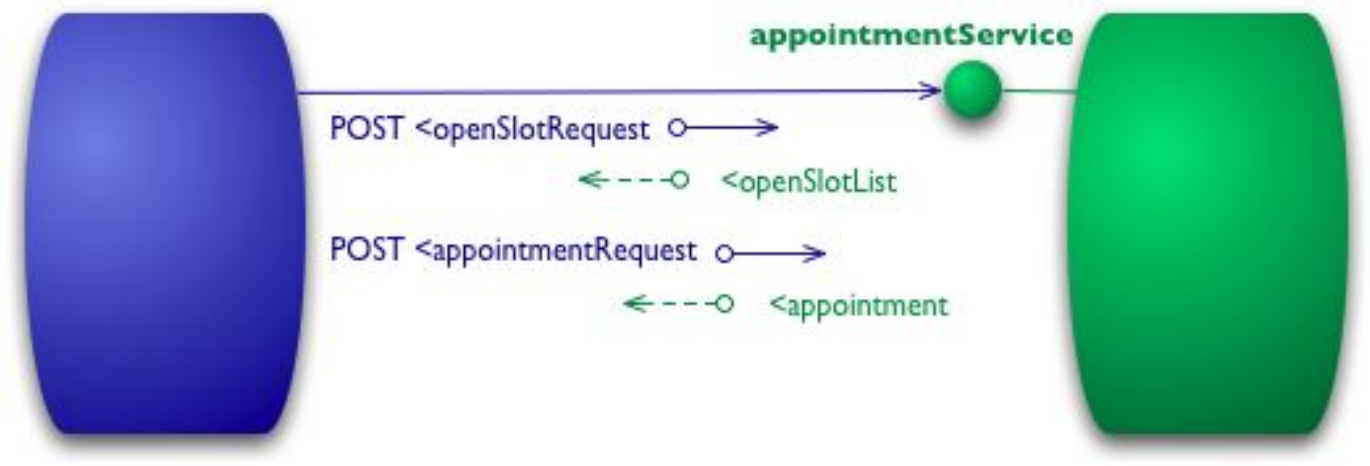


https://martinfowler.com/articles/richardsonMaturityModel.html

Уровень 0

Использование HTTP как транспортного протокола.

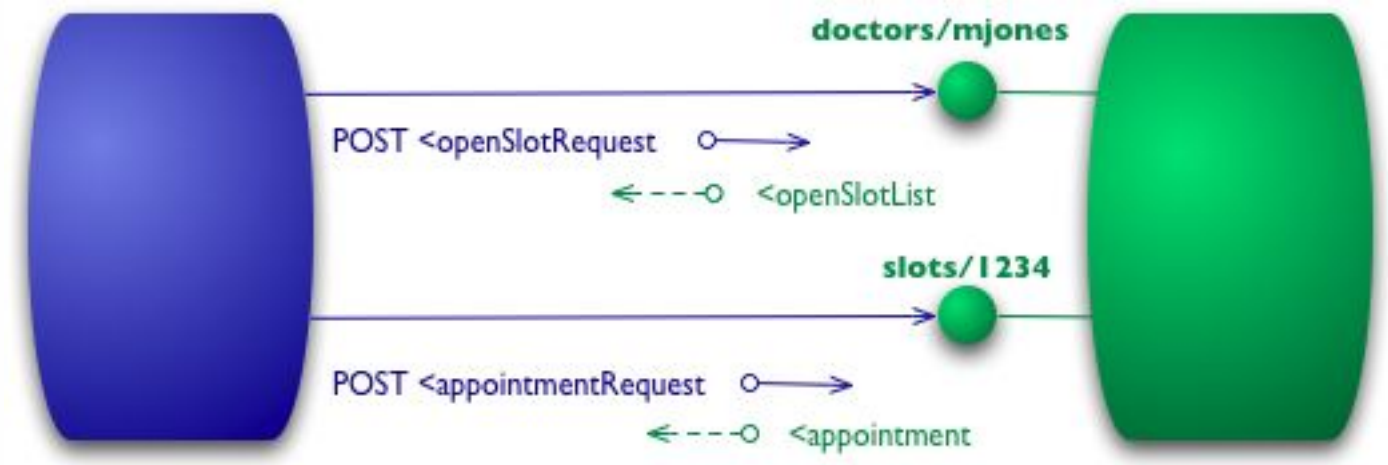
Все запросы идут на 1 endpoint и одним методом



https://martinfowler.com/articles/richardsonMaturityModel.html

Уровень 1

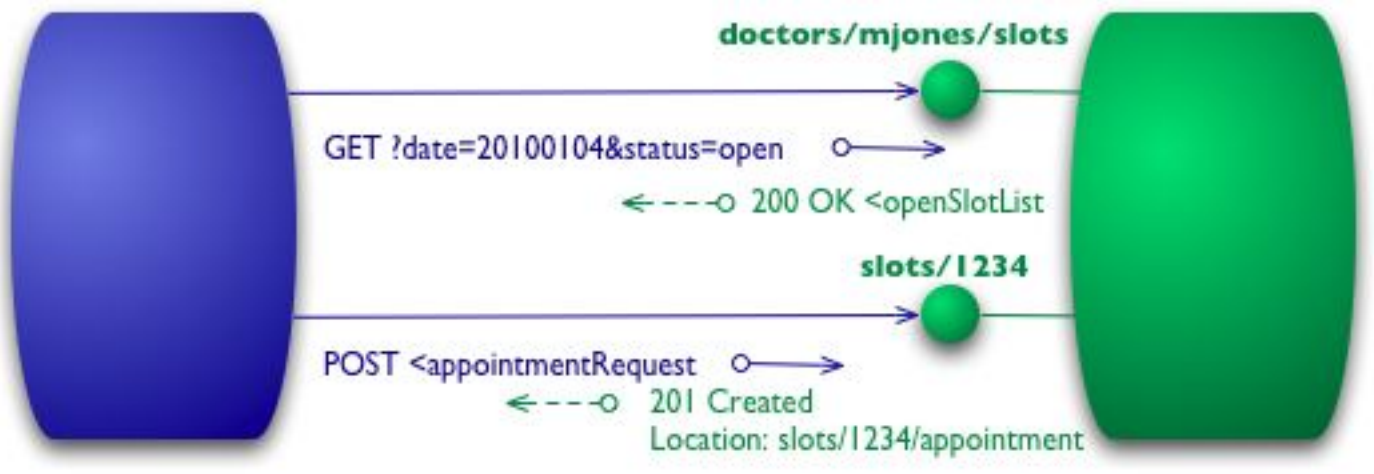
Использование разных ресурсов для разных сущностей, но с одним методом



https://martinfowler.com/articles/richardsonMaturityModel.html

Уровень 2

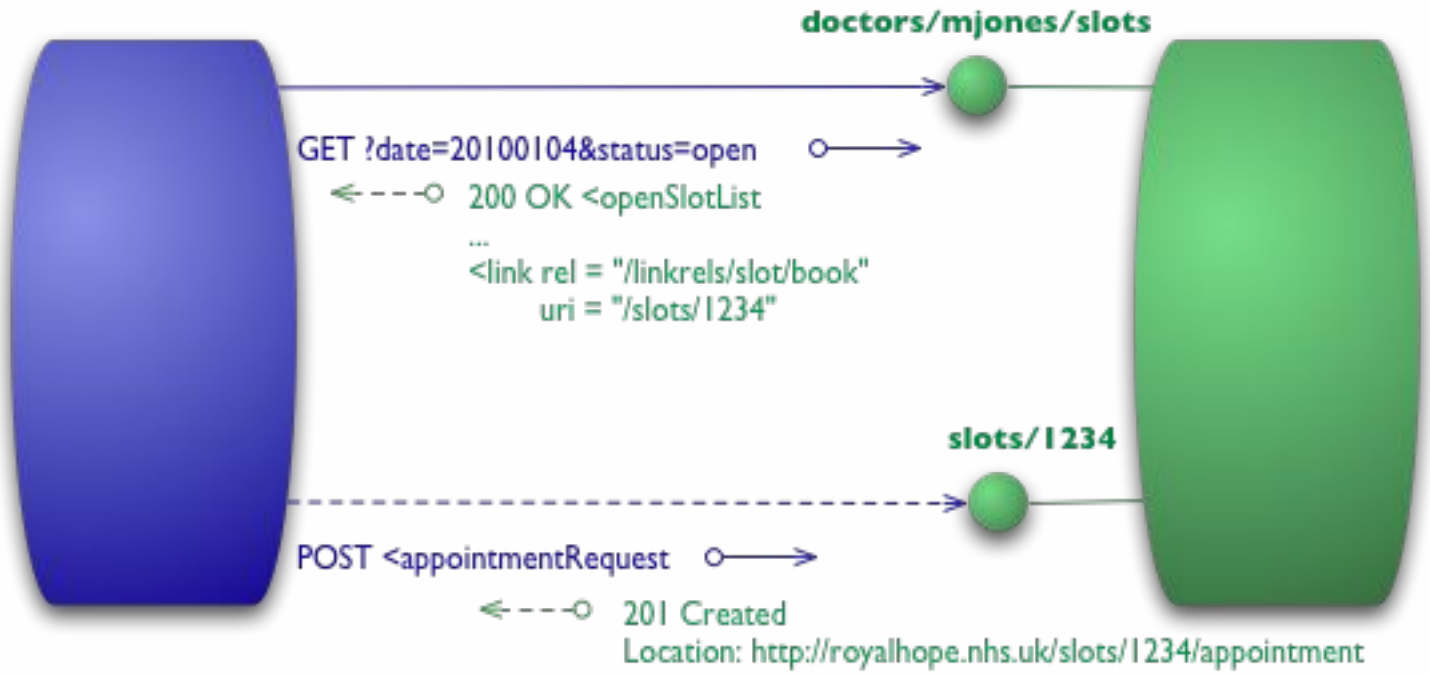
Использование разных ресурсов для разных сущностей.
Использование разных методов



https://martinfowler.com/articles/richardsonMaturityModel.html

Уровень 3

Использование разных ресурсов для разных сущностей.
Использование разных методов
Использование HATEOS (hypertext as the engine of app state)



https://martinfowler.com/articles/richardsonMaturityModel.html

HATEOS

HATEOS - hypertext as the engine of application state – это ограничения к REST API

Клиенту не обязательно знать, где находится тот или иной ресурс. Важно знать, где находится корневой, и переходя по ссылкам в API можно найти всю необходимую информацию.

Не должно быть какой-то предопределенных иерархий.

Клиенту должен выбирать исходя из того, что для выбора предоставляет сервер.

Таким образом достигается отделение серверной логики от клиентской.

https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

HATEOS

```
GET /accounts/12345 HTTP/1.1
Host: bank.example.com
Accept: application/vnd.acme.account+json
...
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.acme.account+json
Content-Length: ...

{
    "account": {
        "account_number": 12345,
        "balance": {
            "currency": "usd",
            "value": 100.00
        },
        "links": {
            "deposit": "/accounts/12345/deposit",
            "withdraw": "/accounts/12345/withdraw",
            "transfer": "/accounts/12345/transfer",
            "close": "/accounts/12345/close"
        }
    }
}
```

https://en.wikipedia.org/wiki/HATEOAS

HATEOS

Ссылки при это отражают текущее СОСТОЯНИЕ ресурса. Например, при отрицательном балансе доступно только пополнение

```
HTTP/1.1 200 OK
Content-Type: application/vnd.acme.account+json
Content-Length: ...

{
    "account": {
        "account_number": 12345,
        "balance": {
            "currency": "usd",
            "value": -25.00
        },
        "links": {
            "deposit": "/accounts/12345/deposit"
        }
    }
}
```

https://en.wikipedia.org/wiki/Hypertext_Application_Language

REST

Важно понимать, что изначально идея Филдинга заключалась в том, что дать возможность сервисам взаимодействовать друг с другом также как взаимодействует обычный человеку с HTML сайтом. Человек знает основную ссылку, а дальше уже переходит ориентируясь на “форму и интерфейсе”.

Для машины интерфейсом должен был стать hyper-media engine и основные усилия должны были быть предприняты в этом направлении.

REST не про то, как называются ресурсы и коллекции, и JSON . Более того, описание API – это НЕ REST путь. REST путь – возможность ходить по API без необходимости знать точную спецификацию URL и формата данных.

https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

От REST к RESTful

Жизнь оказалась немного сложнее. И то, что удобно для человека, не очень удобно для машин. И как показала практика, идея использовать HTTP и его возможности для межсервисного взаимодействия оказалась популярной и удачной. Действительно множество инструментов, написанных для HTTP и встроенных в HTTP возможностей были удобны при создании API: человекочитаемые URI, механизмы кэширования, application/type, передача файлов и т.д. и т.п.

С другой стороны идея Hypermedia для межсервисного взаимодействия поддержки не нашла.

Все это привело к тому, что появились RESTful сервисы, которые адаптировали часть идей Филдинга, но не все, и по праву REST сервисами считаться не могут.

От REST к RESTful

Как восприняли REST обычные разработчики?

Они его восприняли как идею маппинга объектной парадигмы в API.

Ресурс – это объект с его атрибутами, а коллекция – класс. Что дало возможность как раз-таки быстро и просто создавать CRUD интерфейсы.

Т.е есть табличка article => ArticleModel => Article Serializer (DTO) => /articles/

Запись в табличке article => ArticleInstance(id=42) => /articles/42

Что конечно же НЕ является идеей Филдинга, но это то во что по сути превратился RESTful дизайн.

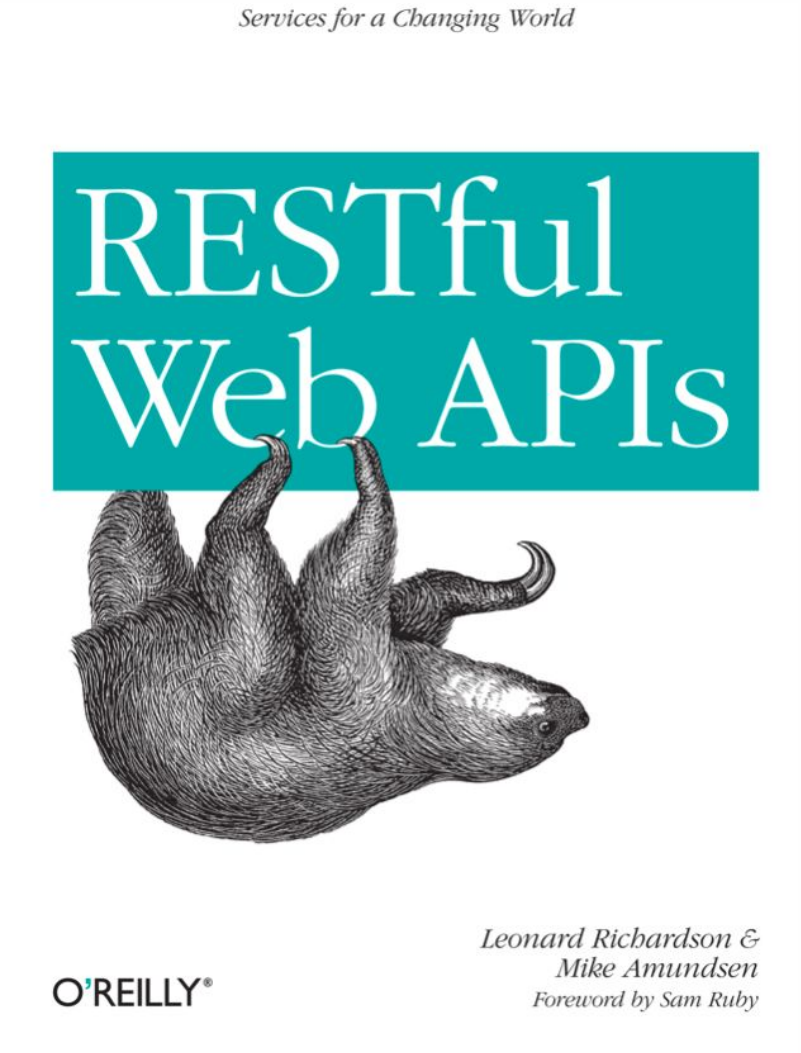
Обработка состояния

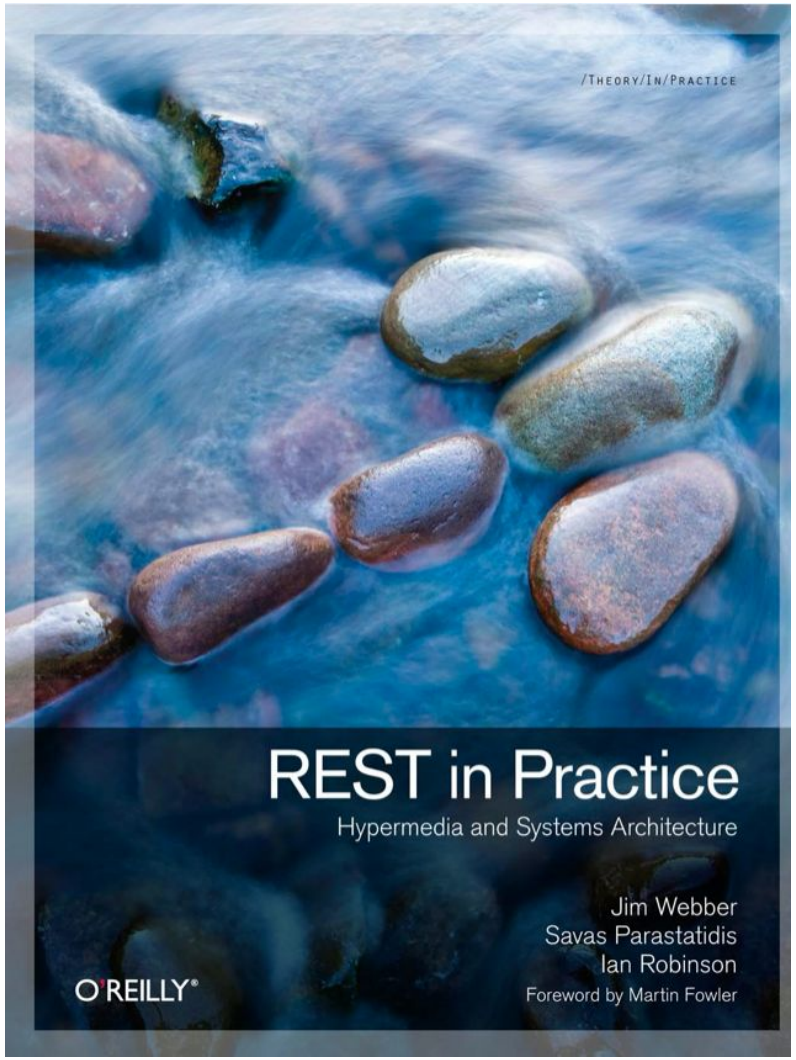
Протокол взаимодействия между клиентом и сервером требует соблюдения следующего условия: в период между запросами клиента никакая информация о состоянии клиента на сервере не хранится (Stateless protocol или «протокол без сохранения состояния»). Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Состояние сессии при этом сохраняется на стороне клиента. Информация о состоянии сессии может быть передана сервером какому-либо другому сервису (например, в службу базы данных) для поддержания устойчивого состояния, например, на период установления аутентификации. Клиент инициирует отправку запросов, когда он готов (возникает необходимость) перейти в новое состояние.

Во время обработки клиентских запросов считается, что клиент находится в переходном состоянии. Каждое отдельное состояние приложения представлено связями, которые могут быть задействованы при следующем обращении клиента.

(wikipedia)

От REST к RESTful





02

Паттерны RESTful API

RESTful

RESTful является подходом, и никак не говорит о том, как решать обычные задачи, с которыми сталкиваются разработчики и архитекторы сервисов:

- Фильтрация и querying
- Работа с иерархией ресурсов в рамках одного запроса
- Пагинация
- Ссылки на другие сущности
- Асинхронные запросы
- Батч запросы
- Формат сообщений об ошибках
- Семантику HTTP кодов в приложении к конкретным ситуациям

Что родило несколько разноплановых стандартов и спецификаций. И паттернов – того, как обычно делают.

JSON-API

JSON-API – спецификация на RESTful API на JSON.

Эта спецификация использует максимально возможности HTTP и отвечает на вопросы:

- Формат ответа
- Фильтрация, quering, пагинация
- Ссылки на связанные ресурсы
- Обновление, удаление ресурсов
- Формат ошибок
- Семантика HTTP кодов

https://jsonapi.org/

ODATA protocol

ODATA – очень развернутый и максимально REST-like протокол, который специфицирует все что только можно.

https://www.odata.org/

Получение ресурса

```
GET /articles/15 HTTP/1.1

200 OK
{
   "id": 15,
   "name": "Very important article!"
   "author": {
      "id": 42
   }
}
```

Получение ресурса вместе с мета-информацией

```
GET /articles/1 HTTP/1.1

200 OK
Content-Type: application/vnd.api+json
{
  "type": "articles",
  "id": "1",
  "attributes": {
    "title": "Rails is Omakase"
  },
  "relationships": {
    "author": {
      "links": {
        "self": "http://example.com/articles/1/relationships/author",
        "related": "http://example.com/articles/1/author"
      },
      "data": { "type": "people", "id": "9" }
    }
  },
  "links": {
    "self": "http://example.com/articles/1"
  }
}
```

Получение ресурса вместе с мета-информацией

```
GET /People('russellwhyte')

200 OK
{
   "@odata.context": "/$metadata#People/$entity",
   "@odata.id": "/People('russellwhyte')",
   "@odata.etag": "W/"08D1694BF26D2BC9"",
   "@odata.editLink": "/People('russellwhyte')",
   "UserName": "russellwhyte",
   "FirstName": "Russell",
   "LastName": "Whyte",
   "Emails": ["Russell@example.com"],
   "AddressInfo": [
      {"Address": "187 Suffolk Ln."}
   ]
}
```

Обновление ресурса

```
PATCH /articles/15 HTTP/1.1
{
    "name": "New name"
}


200 OK
{
    "id": 15,
    "name": New name"
    "author": {
        "id": 42
    }
}
```

PUT vs PATCH vs POST

HTTP отделяет семантику PUT, PATCH и POST запросов

- **PUT** – заменить ресурс
  В запрос должно прийти ресурс со всеми полями
- **PATCH** – частичное изменение ресурса
- **POST** – создание ресурса

Создание ресурса

```
POST /articles HTTP/1.1
{
   "name": "Very important article!"
   "author": {
      "id": 42
   }
}
```

```
201 CREATED
{
   "id": 15,
   "name": "Very important article!"
   "author": {
      "id": 42
   }
}
```

Удаление ресурса

DELETE /articles/1 HTTP/1.1

204 NO CONTENT

Получение списка ресурсов

```
GET /articles HTTP/1.1

200 OK
[
   {
      "id": 1,
      "name": "The importance of metalinguistic"
      "author": {
         "id": 27
      }
   },
   {
      "id": 2,
      "name": "The Foo of Bar In Quux"
      "author": {
         "id": 34
      }
   }
]
```

Фильтрация и quering

Часто хочется получить какую-то часть коллекции, а не всю.
Для этого часто используют фильтры и язык запросов

Обычный query-string
GET /cars?color=blue&type=sedan&doors=4
POST /cars/filter {“color”: “blue”, “doors”: 4}

ODATA:
GET /Airports?$filter=contains(Location/Address, 'San Francisco’)
GET /Suppliers?$filter=Address/City eq 'Redmond’
GET /Products?$filter=Price le 3.5 or Price gt 200
GET /users?$filter=startswith(givenName,'J’)

Фильтры как ресурсы
POST /filters/
GET /api/users/?filterId=1234-abcd

Partial response

Иногда ресурсы бывают многословны и все поля с сервера передавать не надо для оптимизации, как со стороны сервера, так и клиента.

Обычное поле fields
GET /api/v1/products?fields=id,name,price
GET /api/v1/products?fields=id,name,price,delivery(id,shop_id)

ODATA
GET /Hotels?$select=HotelName, Address, Rooms/Type, Rooms/BaseRate

JSON-API
GET /articles?include=author&fields[articles]=title,body&fields[people]=name

Expand/Include in response

Иногда хочется в ресурсе получить дополнительную информацию по связанным объектам.

JSON-API
GET /articles/1?include=comments
GET /articles/1?include=author,comments.author

ODATA
GET /Categories?$expand=Products
GET /Categories?$expand=Products/Supplier
GET /Categories?$expand=Products&$select=Name,Products/Name

Пагинация

Есть несколько типов пагинации

По способу указания места и размера страницы

- Постраничная
- Offset-limit
- Cursor-based

И есть несколько способов передавать данные

- В query параметрах
- В X- заголовках

https://www.citusdata.com/blog/2016/03/30/five-ways-to-paginate/

Постраничная пагинация

Постраничная пагинация – это с клиента приходит только номер странички и иногда количество элементов на страничку

GET /Products/?page=2&page_size=50

Или например в заголовках:

GET /Products/
X-Paging-Page: 2
X-Paging-Page-Size: 50

JSON-API
GET /articles?page[size]=50&page[number]=3

Offset-limit пагинация

Offset-limit пагинация – страничка задается с помощью offset/limit.
Offset- с какого элемента по номеру, limit –размер

GET /Products/?offset=20&limit=50

Или например в заголовках:

GET /Products/
X-Paging-Offset: 20
X-Paging-Limit: 50

JSON-API
GET /articles?page[limit]=50&page[offset]=30

ODATA
GET /articles?$top=5&$skip=1

Cursor пагинация

Cursor пагинация – следующую страничку получаем с помощью курсора, который указывает на текущий элемент в пагинации. При этом в ответе обычно еще и присутствуют ссылки на следующую и предыдущую страничку.

GET /Products/?after=aheD5_x&limit=50
GET /Products/?before=aheD5_x&limit=50

JSON-API
GET /articles?page[after]=aheD5_x&page[size]=30
GET /articles?page[befor]=aheD5_x&page[size]=30

https://medium.com/@meganchang_96378/how-to-implement-cursor-pagination-like-a-pro-513140b65f32

Пример cursor пагинация. Twitter

https://api.twitter.com/1.1/search/tweets.json?q=php&since_id=2401261998405100 0&max_id=250126199840518145&result_type=recent&count=10


```
"search_metadata":
{
   "max_id": 250126199840518145,
   "since_id": 240126199840510 00,
   "next_results": "?max_id=249279667666817023&count=10&include_entities=1",
   "count": 10,
   "since_id_str": "240126199840510 00",
   "max_id_str": "250126199840518145"
}
```


https://www.sitepoint.com/paginating-real-time-data-cursor-based-pagination/

Пример cursor пагинация. Facebook


```
{
   "data": [  …. ],
   "paging": {
      "cursors": {
         "after": "MTAxNTExOTQ1MjAwNzI5NDE=",
         "before": "NDMyNzQyODI3OTQw"
      },
      "previous": "https://graph.facebook.com/me/albums?limit=25&before=NDMyNzQyODI3OTQw",
      "next": "https://graph.facebook.com/me/albums?limit=25&after=MTAxNTExOTQ1MjAwNzI5NDE="
   }
}
```


https://www.sitepoint.com/paginating-real-time-data-cursor-based-pagination/

Cursor пагинация vs Offset-Limit/Page пагинации

Offset-Limit/Page пагинация

- Плохо работает для больших данных. Чтобы получить данные базе придется перечитать все offset записи.
- Плохо работает, если данные в момент запроса добавляются. Что может привести к наличию дубликатов в момент чтения и ошибкам в разработке.

Cursor пагинация

- Хорошо работает для больших датасетов
- Отсутствие дубликатов
- В случае нечитаемых курсоров, приходится последовательно читать, чтобы получить N-ую страничку.

Асинхронные запросы в API

Асинхронные запросы – возможность сделать очень долгий запрос, или запрос, которые не может быть выполнен прямо сейчас (например, из-за ограничений в троттлинге), не дожидаясь непосредственно ответа.

Чаще всего такие запросы возвращают id задачи или url, по которому можно забрать результат.

Асинхронные запросы в API

POST /async/send_email/

```
{
   "subject": "YoLo",
   "email": "user@example.com"
}
```

202 ACCEPTED
Location: /jobs/12313420000123101

```
{
   "job_id": "12313420000123101",
   "link": "/jobs/12313420000123101"
}
```

Асинхронные запросы в API

GET /jobs/1231342000012310 1

200 OK
```
{
   "status": "PENDING",
   "results": null
}
```

И через некоторое время результаты появляются

GET /jobs/1231342000012310 1

200 OK
```
{
   "status": "COMPLETED",
   "results": "{\"id\": 42, \"status\": \"SENT\"}"
}
```

Webhook-и

Webhook (или callback) – еще один асинхронный механизм уведомлений со стороны сервера о событиях.

Пользователь регистрирует подписку на определенные события и указывает на какой url должен сходить сервер, чтобы уведомить о его наступлении.

Webhook на примере target.my.com

https://target.my.com/doc/api/ru/resource/Subscriptions

```
POST https://target.my.com/api/v3/subscription.json

{
    "resource": "BANNER",
    "callback_url": "https://domain.ru/mt/callback/"
}
```


```
{
    "id": "07c0810ac51c47c98e001b1e91c94ba4",
    "resource_id": 1,
    "resource": "OKLEADAD",
    "callback_url": "https://domain.ru/mt/callback/",
    "created": "2019-06-02 18:23:29.797499",
    "data": {
        "id": 1,
        "banner_id": 3368796,
        "campaign_id": 941123,
        "created_time": "2017-11-16 18:04:09",
        "form_id": 457,
        "form_name": "Tinkoff Test Lead Ads",
        "user_birthday": "1990-12-24",
        "user_email": "diway@mail.ru",
        "user_fullname": "Артём Антонов",
        "user_geo": "Лондон",
        "user_phone": "+7 9153139850",
        "user_questions": [
            ["Выберите тип карты", "Выберите вариант"],
            ["Выберите платежную систему", "Выберите вариант"]
        ],
        "user_sex": "unknown"
    }
}
```

Webhook на примере github.com

https://developer.github.com/webhooks/

```
{
  "name": "web",
  "active": true,
  "events": [
    "push",
    "pull_request"
  ],
  "config": {
    "url": "https://example.com/webhook",
    "content_type": "json",
    "insecure_ssl": "0"
  }
}
```

```
Status: 201 Created
Location: https://api.github.com/repos/octocat/Hello-World/hooks/12345678
```

```
{
  "type": "Repository",
  "id": 12345678,
  "name": "web",
  "active": true,
  "events": [
    "push",
    "pull_request"
  ],
  "config": {
    "content_type": "json",
    "insecure_ssl": "0",
    "url": "https://example.com/webhook"
  },
  "updated_at": "2019-06-03T00:57:16Z",
  "created_at": "2019-06-03T00:57:16Z",
  "url": "https://api.github.com/repos/octocat/Hello-World/hooks/12345678",
  "test_url": "https://api.github.com/repos/octocat/Hello-World/hooks/12345678/test",
  "ping_url": "https://api.github.com/repos/octocat/Hello-World/hooks/12345678/pings",
  "last_response": {
    "code": null,
    "status": "unused",
    "message": null
  }
}
```

Массовое изменение

Иногда хочется изменить не один объект, а множество. Чаще всего для этого создают специальные методы по массовому изменению.

Например, https://target.my.com/doc/api/ru/resource/BannerMassAction

```
POST https://target.my.com/api/v2/banners/mass_action.json

[
    {
        "id": 123456,
        "status": "active",
    },
    {
        "id": 234567,
        "status": "blocked"
    },
]
```

Batch запросы

Иногда хочется сделать несколько запросов одновременно, но каждый раз при этом не ходить на сервер.

Для этого существуют batch запросы – несколько запросов за один.

Batch запросы в Facebook API

https://developers.facebook.com/docs/graph-api/making-multiple-requests/

```
curl \
-F 'access_token=...' \
-F 'batch=[
  {
    "method":"POST",
    "name":"create-ad",
    "relative_url":"11077200629332/ads",
    "body":"ads=%5B%7B%22name%22%3A%22test_ad%22%2C%22billing_entity_id%22%3A111200774273%7D
  },
  {
    "method":"GET",
    "relative_url":"?ids={result=create-ad:$.data.*.id}"
  }
]' \
https://graph.facebook.com
```

```
[
    { "code": 200,
      "headers": [
          { "name":"Content-Type",
            "value":"text/javascript; charset=UTF-8"}
       ],
      "body":"{\"id\":\"…\"}"
    },
    { "code": 200,
      "headers": [
          { "name":"Content-Type",
            "value":"text/javascript; charset=UTF-8"
          },
          { "name":"ETag",
            "value": "…"
          }
      ],
      "body": "{\"data\": [{…}]}
    }
]
```

03

IDL RESTful сервисов

JSON Schema

Для того, чтобы валидировать и описывать формат JSONa, у которого изначально такой схемы не было, изобрели JSON-Schema - это формат описания схемы JSON-а.

https://json-schema.org/understanding-json-schema/

https://json-schema.org/

JSON Schema

```
{
        "name": "users.get",
        "description": "Returns detailed information on users.",
        "open": true,
        "parameters": [
          {
            "name": "user_ids",
            "description": "User IDs or screen names ('screen_name'). By
            "type": "array",
            "items": {
              "type": "string"
            },
            "maxItems": 1000
          },
          {
            "name": "fields",
            "description": "Profile fields to return. Sample values: 'ni
            'can_write_private_message', 'can_see_all_posts', 'can_post'
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          {
            "name": "name_case",
            "description": "Case for declension of user name and surname
            "type": "string"
          }
        ],
        "responses": {
          "response": {
            "$ref": "responses.json#/definitions/users_get_response"
          }
        }
      }
```

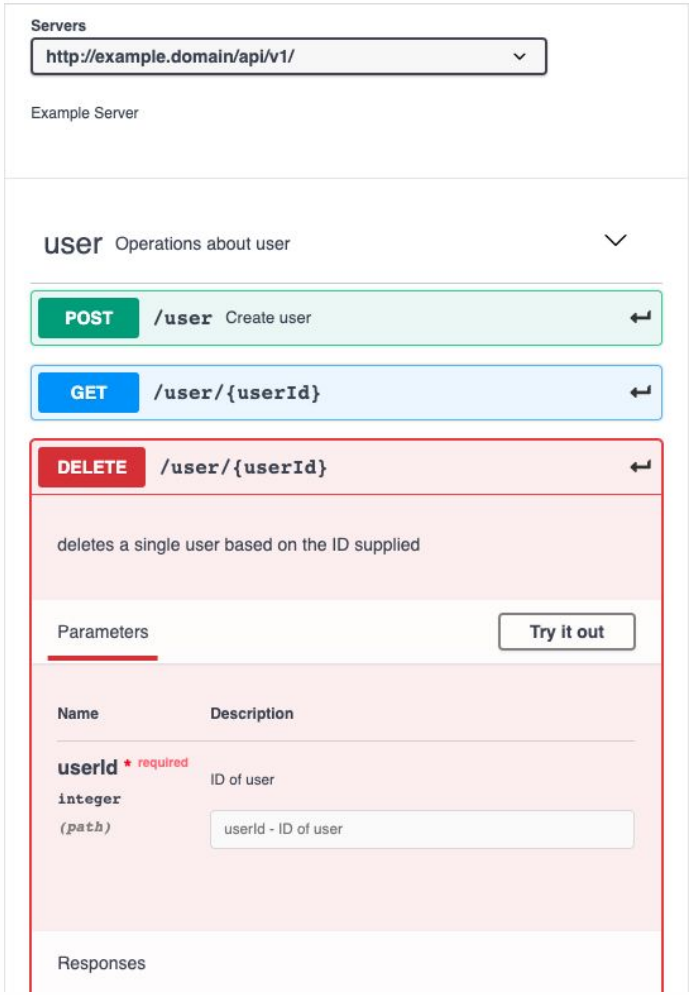
IDL для RESTful API

Для RESTful есть язык описания интерфейса – OpenAPI, бывший Swagger.

На текущий момент есть две версии 2 и 3.

IDL для RESTful API

```yaml
  - description: Example Server
    url: http://example.domain/api/v1/
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/otus55/users/1.0.0
tags:
  - name: user
    description: Operations about user
paths:
  /user:
    post:
      tags:
        - user
      summary: Create user
      description: This can only be done by the logged in user.
      operationId: createUser
      responses:
        default:
          description: successful operation
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
            examples:
              sample-user:
                summary: Example
                value:
                  username: johndoe589
                  firstName: John
                  lastName: Doe
                  email: bestjohn@doe.com
                  phone: '+71002003040'
        description: Created user object
        required: true
  /user/{userId}:
    parameters:
    - name: userId
      in: path
```



https://app.swaggerhub.com/apis/otus55/users/1.0.0

Спасибо
за внимание!

https://otus.ru/polls/36001/