Topic 4: Specifying and using RESTful APIs

Dr. Daniel Yellin

Overview

- Safe and Idempotent APIs
- API Keys
- OpenAPI Specifications
- HATEOS
- REST architectural constraints

Safe and Idempotent APIs

Repeating REST invocation on a service

What if the API endpoint does not respond to the request?

- Should the client keep on trying to call the API again and again?
- What if the provider is just a bit slow today and eventually sees all the client invocation?
- What problems can arise?

Does Noa get added to the list? How many times?

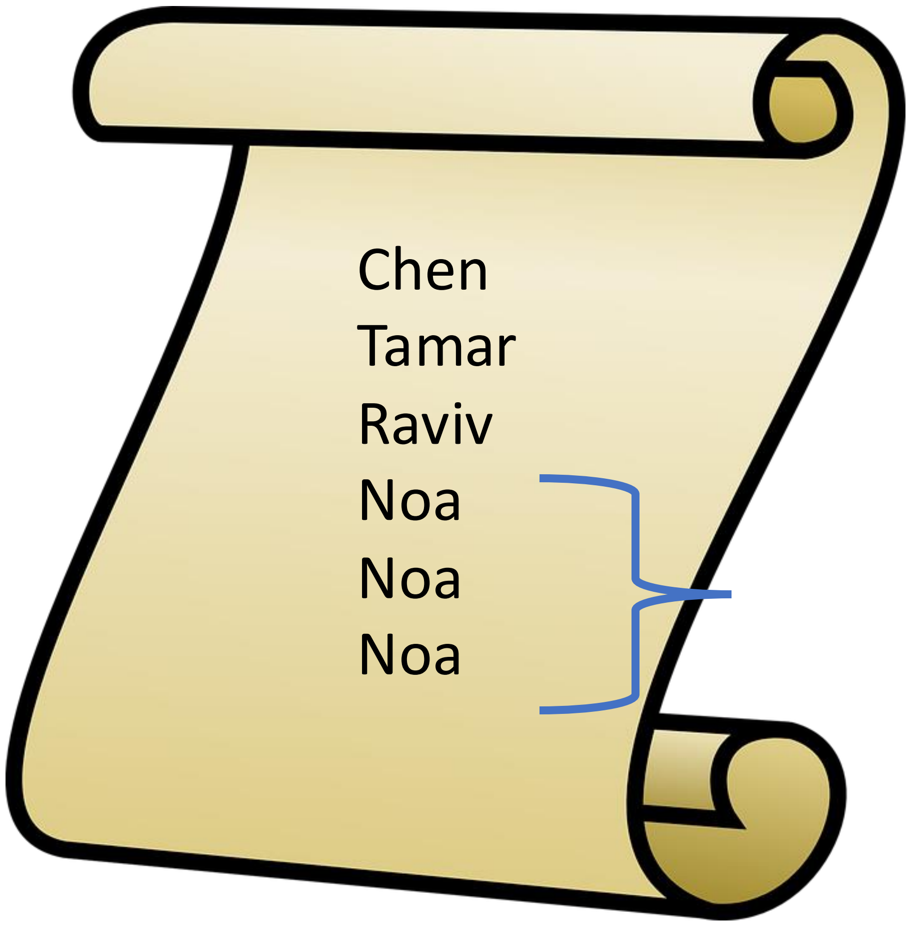
“Add Noa to the list”

“Did you hear me? I said Add Noa to the list”

“You are not answering me! I said Add Noa to the list”

“Alright”
“Yes I heard you”
“Ok, Ok!"

The list



Chen
Tamar
Raviv
Noa
Noa
Noa

Noa was only supposed to be added once.

Safe HTTP methods

Safe HTTP methods *do not alter the server state.*

What methods do you think are safe?

Safe HTTP methods

- **Safe** HTTP methods *do not alter the server state.* Hence only read-only operations are safe.
  - The HTTP RFC defines the following methods to be safe: GET, HEAD, OPTIONS and TRACE.
- In practice it is often not possible to implement safe methods in a way that they do not alter *any* server state.
  - For example, a GET request might create log or audit messages, or trigger a cache refresh on the server.
  - In general, safe methods should not alter any state *important* to the client.

HTTP operation	Safe?
GET	YES
POST	NO
PUT	NO
DELETE	NO
PATCH	NO
HEAD, OPTIONS, TRACE	YES

Idempotent HTTP methods

Idempotency means that multiple identical requests will have the same effect *on the resource* as if issued just once. It does not matter if a request is sent once or multiple times.

What methods do you think are idempotent?

Idempotent HTTP methods

- **Idempotency** means that multiple identical requests will have the same effect *on the resource* as if issued just once. It does not matter if a request is sent once or multiple times. The following HTTP methods are idempotent:
    - GET, HEAD, OPTIONS, TRACE, PUT and DELETE. All safe HTTP methods are idempotent but PUT and DELETE are idempotent but not safe.
- Note that idempotency does not mean that the server has to respond in the same way on each request.
    - A DELETE request will give a different response the second time than it did on the first time.

HTTP operation	Safe?	Idempotent ?
GET	YES	YES
POST	NO	NO
PUT	NO	YES
DELETE	NO	YES
PATCH	NO	NO
HEAD, OPTIONS, TRACE	YES	YES

Be careful with these ones

Idempotent APIs

Idempotency is a positive feature of an API because it can make an API more *fault-tolerant.*

- Assume there is an issue on the client and requests are send multiple times. As long as idempotent operations are used this will cause no problems on the server side.
- PUT is idempotent – if I replace the same resource multiple times with the same value, it does not change the state of the resource on the server.
- POST is not idempotent – if I ask for a resource to be created two times, it will be created two times (usually given different IDs).

Why isn’t Patch idempotent?

PATCH is defined as a *partial update*, and partial updates can depend on the *current state* of the resource.

- If the update depends on the resource's current value, then repeating the request can keep changing the resource.
- **Example:** PATCH /account/123

  { "balance": "+10" }

This causes the balance to increase by 10 on each invocation.

This is due to the way that PATCH is defined by the HTTP standard. A PATCH request is interpreted as a set of instructions on how a resource is to be updated.

This is different from a a PUT request, where the values of the PUT request modify the current values of the resource.

API Keys

Who is using your API?

Are API calls anonymous?

- How does the API provider know who is calling the API?
- How does the API provider know if the caller is allowed to access the information she is requesting?
- How does the API provider know who to charge for her services?

API Keys

REST APIs often require a parameter which is the *API Key*

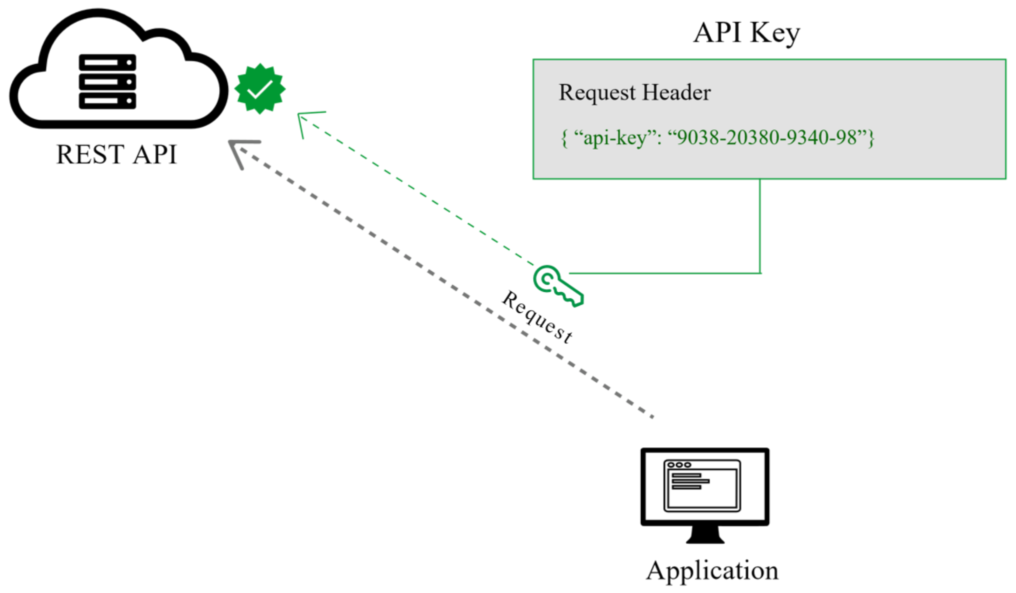
- One acquires an API key from the API provider, usually by registering.
- On all invocations of the API, you must provide your specific API Key.

There are three main reasons for API keys

1. Authentication
    - Including the API Key indicates that you are an authorized user of the API. It prevents malicious or excessive usage of the API by unauthorized users.
2. Metering and billing
    - To obtain the API key, the provider may require the consumer to provide a means for payment, e.g., a credit card. This allows the provider to bill you for usage of the API, based upon their terms of usage
    - Payment plans are often tiered: usage below a certain # of calls per day may be free, and higher usage tiers will cost more.
3. Access control
    - The API key may govern which resources you have access to. For instance, if the API allows you to create your own resources, only users providing your API key can access those resources.

API keys and security

You need the API key in order to invoke a service. Where do you store the key?



In your code

Anyone who has access to your code can see your private key!

Anytime you change your key you have to change your code.

Storing the key in your Dockerfile is a little better, but still has security issues.

```
def get_nutritional_info(ingredient):
    """
    Get nutritional information for an ingredient using API Ninjas
    """
    try:
        query = ingredient
        api_url = 'https://api.apininjas.com/
                        v1/nutrition?query={}'.format(query)
        response = requests.get(api_url, headers={'X-Api-Key':
                                                                    sp$$5fg55RAn&88})

        if response.status_code == 200:
            data = response.json()
            # the data should be a json list with one element, which is a json
structure containing the nutritional info
            if data and len(data) > 0:
                return data[0]
            return {}
        else:
            # API error handling
            return {}
```

Use a secrets management system

- Stores secrets, such as API keys, in a secrets repository
  - Secrets are encrypted
  - Users are authenticated before gaining access
  - Policies dictate which users can access which secrets
  - Provides other features, such as facilitating *rotation* of encryption keys
- Example, Hashicorp Vault

We will not use a secrets management system in this course.

OpenAPI Specifications

Formalizing REST-like interfaces

OpenAPI Specifications

While REST is an *architectural style* and provides guidelines on building APIs, it lacks a mechanism for precisely specifying APIs. OpenAPI is a *formal language* for specifying REST-like APIs. Advantages of using OpenAPI is:

- **Precise** (unambiguous) **semantics**
- **Machine readable**
  - Many tools that can process OpenAPI specs for testing, code generation, ... See OpenAPI tools.
- **Human readable**

However, OpenAPI is not REST. For example, it hardwires URLs for server APIs.

OpenAPI specification

An OpenAPI specification defines:

- *Endpoints:* the network paths (URLs) of the service. Such as http:/cats.com/library:5000
- *Operations:* The *requests* supported by each endpoint (e.g., GET, POST, …)
- *Parameters*: The input required for each operation, which can be passed in the path, query string, headers, or cookies.
- *Responses*: the responses codes and bodies that can be returned from an operation.
- *Schemas*: Reusable definitions of data models for request and response bodies.
- *Authentication methods*: Details on how to authenticate with the API.
- *Documentation* and *metadata* about the API

An OpenAPI spec for toys service (1)

```
{
 "openapi": "3.1.0",
 "info": {
  "title": "Toy Collection API",
  "description": "An API for managing a collection of toys.",
  "version": "1.0.0"
 },
 "servers": [
  {
   "url": "http://localhost:80",
   "description": "Local server"
  }
 ],
```

Meta-data

Location of **base** URLs that provide this service.

OpenAPI spec for toys service: Paths (2)

```
"paths": {
  "/toys": {
    "get": {
      "summary": "List all toys",
        …
    "post": {
      "summary": "Create a new toy",
         …
  "/toys/{id}": {
    "get": {
      "summary": "Get a specific toy by ID",
        …
```

Container for all API endpoints. .

/toys endpoint.   It supports GET and POST operations.

/toys endpint.   It supports GET, PUT and DELETE operations.

OpenAPI spec for toys service: Operations (3)

```
"/toys": {
    "get": {
     "summary": "List all toys",
     "operationId": "listToys",
     "responses": {
      "200": {
       "description": "A list of toys",
       "content": {
        "application/json": {
         "schema": {
          "type": "array",
          "items": {
           "$ref": "#/components/schemas/Toy"
…
},
```

The HTTP GET operation is defined for the /toys path.

Responses lists all the response codes & bodies for this operation. Besides 200, 404 & 500 are supported (not shown).

The content object provides the permissible media types supported by this operation. In this case, “application/json”.

The schema object defines the datatype of the response object. If it is an array, it must define the item type. In this case, the item type refers to the schema defined elsewhere in the spec.

OpenAPI spec for toys service: schemas (4)

```
"components": {
   "schemas": {
      "Toy": {
       "type": "object",
       "properties": {
        "id": {
          "type": "string"
        },
       …
     "features": {
         "type": "array",
         "items": {
             "type": "string"
         },
         "description": "List of toy features"
       }
     },
```

The components object contains definitions for objects to be reused in other parts of the description.

In this case it defines the schema of a toy. It can be referred to by "$ref": "#/components/schemas/Toy"

One can also define those fields of the object that are mandatory (not shown).

Alternative spec for toys service (4b)

```
"components": {
 "schemas": {
  "Toy": {
    "allOf": [
     { "$ref": "#/components/schemas/ToyInput" },
     { "type": "object",
       "properties": {
         "id": { "type": "string", "readOnly": true }  },
      "required": ["id"]
     }
    ]
  },
  ToyInput": {
    "type": "object",
    "properties": {
     "name": { "type": "string" },
     …
     "price": { "type": "number", "format": "float" },
     "features": {
       "type": "array",
       "items": { "type": "string" }
     }
    },
```

The components object contains definitions for objects to be reused in other parts of the description.

In this case it defines the schema of a toy. It can be referred to by “$ref": "#/components/schemas/Toy"

OpenAPI spec for toys service: parameters (5)

```
"/toys/{id}": {
  "get": {
    "summary": "Get a specific toy by ID",
    "operationId": "getToy",
    "parameters": [
      {
        "name": "id",
        "in": "path",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
   "responses": {
    …
```

The GET operation on the /toys/{id} object takes a parameter.

For each parameter you define its name (mandatory), where the parameter is found (mandatory), whether it is required or not, and its schema..

The “in” field can be one of “**path**” (/toys/{id}), “**query**” (/toys?id=xyz), “**querystring**”, “**header**”, or “**cookie**”.

OpenAPI spec for toys service: api-keys(6)

```
“components”:
 “securitySchemes”:
    “ApiKeyAuth: # arbitrary name
      “description”: “API key provided in header”
      “type”: “apiKey”
      “name”: “X-API-KEY” # name of the header
      “in”:” header”  # "header", "query" or "cookie"
```

There are multiple security schemes supported. We demonstate API keys.

No formal standard for API keys so arbitrary name of schema.

Defines the header name for the api-key. X-API-KEY is a common custom header.

Defines where the API key is to be found

See here

OpenAPI pros and cons

Pros

- Language-agnostic
- Wide adoption in the industry
- REST is an architectural *style:* it leaves many things unspecified
    - A standard way of defining the interface
    - A wire protocol for describing communication between endpoints
    - **OpenAPI** standardizes exactly how to write an API
- Proliferation of tools that automatically process an OpenAPI spec

Caveats

- Complexity and "API bloat", especially for large APIs.
- Steep learning curve. Structuring complex schema can be difficult.
- Can lead to over-specification.
- Great for static API structures (paths, parameters, responses) but does not capture dynamic properties of APIs (e.g., dynamic content generation or highly conditional response data).

HATEOAS

Hypermedia as the engine of application state

The uniform interface constraint

Roy Fielding defined 4 parts to building a RESTful interface:

1. Using **Identifiers** to uniquely identify **resources.**
2. Transmitting **representations** of **resources**, not the resources themselves.
3. Transmitting **meta-data** about the representation (**media types**) in self-descriptive messages so that the client need not know in advance what payload the server will respond with (text? JSON? jpeg? png?).
4. Embedding **hypermedia links** in the API response to refer to related resources.   In contrast to returning the content of those resources in the response.

`We covered the first 3 already.  We now will turn our attention to the “most controversial” constraint, the last one.`

GET http://api.domain.com/management/departments/10
which response below is a better one ?

```
{
     "departmentId": 10,
     "departmentName": "Administration",
     "locationId": 1700,
     "managerId": 200,
     "links": [
           {
               "href": "10/employees",
                "rel": "employees",
                 "type" : "GET"
          }
     ]
}
```

```
{
     "departmentId": 10,
     "departmentName": "Administration",
     "locationId": 1700,
     "managerId": 200,
      "employees": [
           { "name:" "Jack", …
              "employeeID": 123456
           }
            { "name:" "Jill", …
              "employeeID": 987654
           }
           … rest of employees listed here
     ]
}
```

Given the request:
GET http://api.domain.com/management/departments/10
which response below is a better one ?

```
{
      "departmentId": 10,
      "departmentName": "Administration",
      "locationId": 1700,
      "managerId": 200,
      "links": [
             {
                 "href": "10/employees",
                  "rel": "employees",
                   "type" : "GET"
            }
      ]
}
```

Both provide core information on the department

```
{
      "departmentId": 10,
      "departmentName": "Administration",
      "locationId": 1700,
      "managerId": 200,
       “employees”: [
             { “name:” “Jack”, …
                “employeeID”: 123456
             }
              { “name:” “Jill”, …
                “employeeID”: 987654
             }
             … rest of employees listed here
      ]
}
```

Given the request:

GET http://api.domain.com/management/departments/10

which response below is a better one ?

```
{
      "departmentId": 10,
      "departmentName": "Administration",
      "locationId": 1700,
      "managerId": 200,
      "links": [
            {
                  "href": "10/employees",
                   "rel": "employees",
                    "type" : "GET"
            }
      ]
}
```

The left one provides a link to retrieve additional information on employees

The right one provides all the additional information on employees

```
{
      "departmentId": 10,
      "departmentName": "Administration",
      "locationId": 1700,
      "managerId": 200,
       "employees": [
            { "name:" "Jack", …
               "employeeID": 123456
            }
             { "name:" "Jill", …
               "employeeID": 987654
            }
             … rest of employees listed here
      ]
}
```

HATEOS

1. Always passing the client references to resources and not having hardwired urls means that the **server controls** the possible client state transitions – the next state of the client is dictated by the set of embedded links sent back to the client.
2. The server can *dynamically* decide which links to send to the client. It may, for example, **not** send a link to retrieve employee data because this client is not authorized to see that data. It may instead decide to send additional links, such as projects this team is working on.
3. The server can change links. Today it may be "10/employees" and tomorrow something else. The motivation is to allow servers and clients to *evolve independently* without hardcoding expectations.

HATEOS

4. The client does not need to know all the URIs in advance. Just the initial URI to contact the server. The server will then provide the client the links as needed.
5. Fielding named this paradigm **Hypermedia As The Engine of Application State**, or **HATEOAS** for short.

Pros and Cons of HATEOS

Pros

- Facilitates longevity of systems
- Minimizes the assumptions between server and client
  - Servers and clients and evolve independently
  - Clients dynamically interpret links received from server, instead of hard coding the interaction
- Good for external APIs where client and server are very loosely coupled

Cons

- Makes clients harder to write
- Verbose interaction between clients and servers
- Bad for tightly coupled internal systems, especially when performance is critical
- No single accepted standard for specifying links

Note that OpenAPI is not compatible with HATEOS

REST Architecture

REST is more than APIs

REST - *REpresentational State Transfer*

- REST is an architectural *style* – guidelines for building robust, long lasting distributed systems with these attributes:
  - **Scalability** of component interactions
  - **Generality** of interfaces
  - **Independence** of deployment of components
  - Interaction **latency** reduced by intermediary components
  - **Security** enforcement
  - **Encapsulation** of legacy systems

Fielding refers to REST as a "Web based" architectural style. Given his authorship of Internet standards (HTTP, URI), it is not surprising that the emphasis is on *distributed hypermedia systems*.

REST - *REpresentational State Transfer*

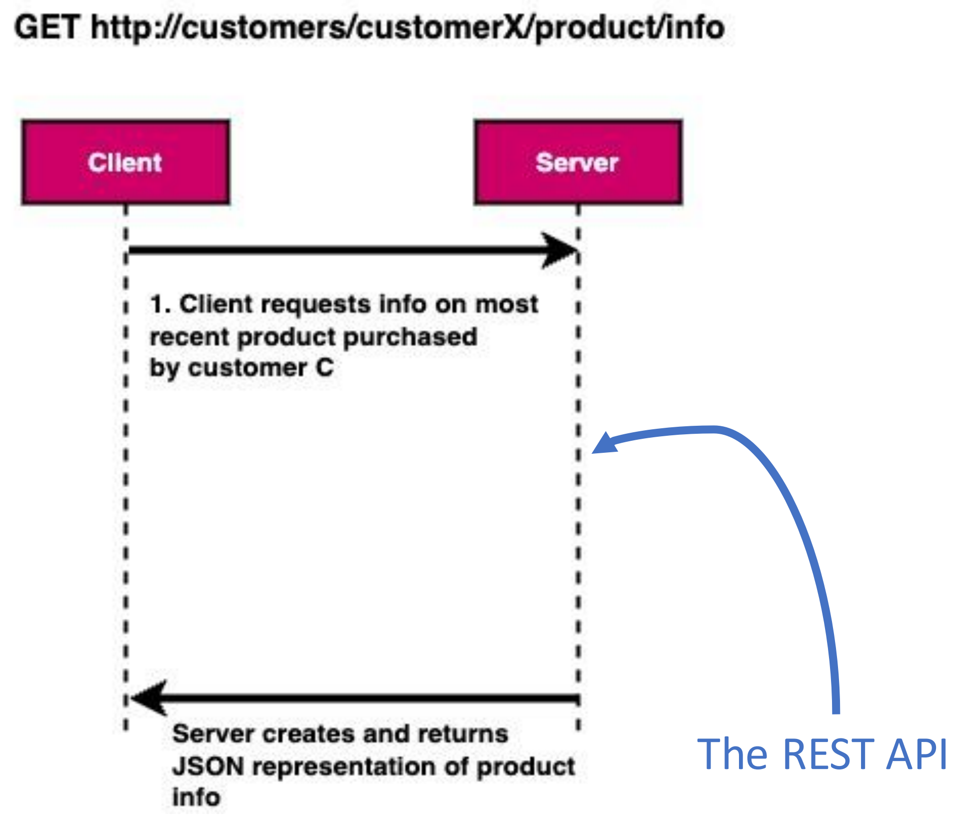
To support these attributes, REST dictates 6 architectural constraints:

1. **Client-server**
2. **Stateless**
3. **Cacheable**
4. **Uniform Interface**
5. **Layered System**
6. **Code on Demand (optional)**

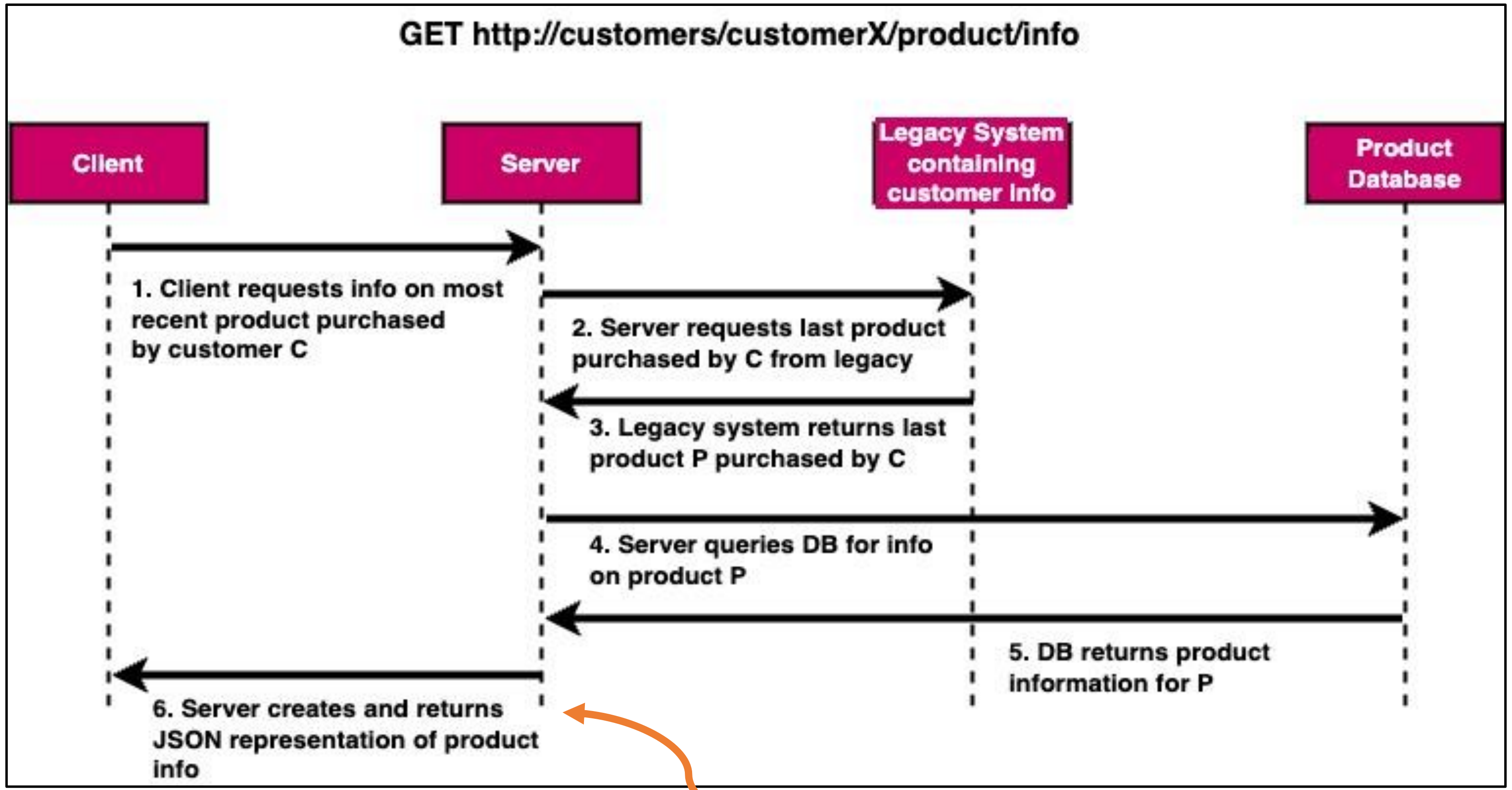
We already discussed (4) Uniform Interface.
We will now describe (1) and (2).

REST Constraint – Client-server

1. **Client-server:** The client and server must be divorced from one another so that each can develop independently & encourages *separation of concerns*.
   - Client need not know details about server implementation and vice-versa.


GET http://customers/customerX/product/info
Client
Server
1. Client requests info on most recent product purchased by customer C
Server creates and returns JSON representation of product info
The REST API

Sequence diagram for a typical REST request


GET http://customers/customerX/product/info
Client
Server
Legacy System containing customer Info
Product Database
1. Client requests info on most recent product purchased by customer C
2. Server requests last product purchased by C from legacy
3. Legacy system returns last product P purchased by C
4. Server queries DB for info on product P
5. DB returns product information for P
6. Server creates and returns JSON representation of product info

The REST API

What the server needs to do to respond to the client's request is hidden from the client – **encapsulation.**

The client only sees the API.

That is goodness!

Change GET Url to match description

REST Constraint – Stateless

2. **Stateless:** The server will not keep any state between client requests.
    - Each request needs to include all the information necessary for processing it (no cookies, no session variables).
    - The stateless constraint reflects a design trade-off. The disadvantage is that it may decrease network performance by increasing the repetitive data (per-interaction overhead) sent in a series of requests.

REST Constraint – Stateless

2. **Stateless:** The server will not keep any state between client requests.
   - Each request needs to include all the information necessary for processing it (no cookies, no session variables).
   - The stateless constraint reflects a design trade-off. The disadvantage is that it may decrease network performance by increasing the repetitive data (per-interaction overhead) sent in a series of requests.

This constraint has the following benefits:

- *Visibility*: all the information is in the request (monitoring).
- *Scalability*: the server does not have to store state (frees state quickly, does not have to locate resources across requests). A load balancer can send request to any server.
- *Reliability*: easy to recover from failures.

Downside to stateless APIs

- More data sent each request
  - Each request must include all the state needed to process the request
- More client-side complexity
  - Clients need to manage session state, workflow state,..
- Harder to implement fine-grained authentication
  - The server must revalidate tokens and compute permissions on every request
- Server-side optimization becomes harder
  - Since the server doesn’t remember anything about the client, it can’t maintain caches per user, optimize based upon past behavior
- Less suited for real-time systems

References

for further exploration

References

- Information hiding and modularity

1. “On the criteria to be used in decomposing systems into modules”, David L. Parnas, CACM, Vol 15, Issue 12, Dec. 1972
2. “The Modular Structure of Complex Systems”, David L. Parnes, Paul C. Clements, David M. Weiss, ” IEEE, vol. SE-1 1, 3, March 1985
3. “A behavioral notion of subtyping”, B. H. Liskov and J. M. Wing, ACM TOPLAS, 16(6), 1994
4. “Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity”, Klaus Ostermann, Paolo G. Giarrusso, Christian Kastner and Tillmann Rendel, ECOOP, 2011
5. “N Degrees of Separation: Multi-Dimensional Separation of Concerns”, P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Proceedings of the 21st International Conference on Software Engineering, 1999

References (2)

- Abstract interfaces
  6. “On Understanding Data Abstraction, Revisited”, W. R. Cook, OOPSLA, 2009
  7. “When should a black box be transparent”, George V. Neville-Neil, CACM, Vol 65, n. 8, August, 2022

References (3)

- General Principles of a good API*
  8. API Design Matters, Michi Henning, CACM, May 2009, vol 52, num 5
  9. How to design a good API and why it matters, Joshua Bloch, Google Video, https://www.youtube.com/watch?v=heh4OeB9A-c
     https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/32713.pdf (Slides)
  10. Design by contracts (Bertrand Meyer) https://www.academia.edu/4903777/Object_Oriented_Software_Construction_SECOND_EDITION
  11. Good API design, bad API design, Roman Kyslyi, https://levelup.gitconnected.com/good-api-design-bad-api-design-2405dcdde24c
  12. Always build with multiple use-cases in mind. "The Rule of Threes", Will Tracz, Confessions of a Used Program Salesman, Addison-Wesley, 1995

*There are additional principles that are specific to programming language design (e.g., classes and methods) but we are concerned primarily about RESTful APIs that are programming language independent.

References (4)

- General Principles of a good API (continued)

13. Solid Principles. Robert Martin, https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
14. The "Single Responsibility Principle", Robert Martin, https://web.archive.org/web/20150202200348/http://www.objectmentor.com/resources/articles/srp.pdf
15. The "Interface Segregation Principle", Robert Martin, https://web.archive.org/web/20150905081110/http://www.objectmentor.com/resources/articles/isp.pdf
16. Design reviews for APIs, "API Design Reviews at Scale", A. Macvean, M. Maly, & J. Daughtry, Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems,May 2016

References (5)

- Extensibility
  17. "The Open Closed Principle", Robert Martin, https://web.archive.org/web/20150207082518/http://www.objectmentor.com/resources/articles/ocp.pdf
  18. Versioning of REST APIs, https://restfulapi.net/versioning/
  19. Versioning of REST APIs, https://www.moesif.com/blog/technical/api-design/Best-Practices-for-Versioning-REST-and-GraphQL-APIs

References (6)

- REST

20. Roy Fielding’s doctoral dissertation defined REST: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
21. There are several good sites that provide explanations of REST and best practices:
22. What is REST, Lokesh Gupta. Recommended overview: https://restfulapi.net/
23. The little book on REST Services, Kenneth Lange, https://www.kennethlange.com/books/The-Little-Book-on-REST-Services.pdf
24. Designing Beautiful REST + JSON APIs,Les Hazlewood: https://www.youtube.com/watch?v=MiOSzpfP1Ww&t=1122s
25. RESTful Service Best Practices, Todd Fredrich, https://raw.githubusercontent.com/tfredrich/RestApiTutorial.com/master/media/RESTful%20Best%20Practices-v1_2.pdf
26. CRUD and REST, https://nordicapis.com/crud-vs-rest-whats-the-difference/
27. Using non-CRUD verbs: https://kennethlange.com/dont-limit-your-rest-api-to-crud-operations/
28. Benefits and disadvantages of HATEOAS, https://engineering.3ap.ch/post/using-hateoas-with-rest/

References (7)

- REST (continued)

29. Good info on HTTP operations, https://www.mscharhag.com/p/rest-api-design
30. Fielding blog on REST & HATEOAS, https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
31. More advanced filtering and sorting, https://www.moesif.com/blog/technical/api-design/REST-API-Design-Filtering-Sorting-and-Pagination
32. REST parameter passing, https://dzone.com/articles/rest-api-design-best-practices-for-parameters-and
33. REST versus SOAP, https://octoperf.com/blog/2018/03/26/soap-vs-rest
34. REST versioning, https://stackoverflow.com/questions/389169/best-practices-for-api-versioning
35. Best practices, https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api#requirements
36. https://www.toptal.com/api-developers/5-golden-rules-for-designing-a-great-web-api
37. https://www.spiceworks.com/tech/cloud/articles/stateful-vs-stateless/

References (8)

38. REST API Design Rulebook, see https://www.oreilly.com/library/view/rest-api-design/9781449317904/ table of contents gives long list of rules
39. "RESTRuler: Towards Automatically Identifying Violations of RESTful Design Rules in Web APIs" https://arxiv.org/pdf/2402.13710.pdf RESTRuler, a Java-based open-source tool that uses static analysis to detect design rule violations in OpenAPI descriptions.

- Related Internet standards.

40. A bit dated and very detailed but good overview of WWW architecture https://www.w3.org/TR/webarch/#identification
41. Latest HTTP standard: https://datatracker.ietf.org/doc/html/rfc9110
42. URI standard syntax. https://www.ietf.org/rfc/rfc3986.txt
43. Wikipedia on URI. https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#cite_note-13
44. Wikipedia on media types. https://en.wikipedia.org/wiki/Media_type
45. How to deal with versions of a resource: https://www.w3.org/2001/tag/doc/alternatives-discovery-20060620.html

References (9)

- API tools:
  - https://curl.haxx.se/ https://httpie.org/
  - https://stedolan.github.io/jq/ https://github.com/antonmedv/fx
  - https://www.postman.com/ https://openapi.tools
  - https://github.com/lornajane/requestbin https://httpbin.org
  - https://github.com/stoplightio/spectral
  - https://stackabuse.com/get-request-query-parameters-with-flask/, https://www.digitalocean.com/community/tutorials/processing-incoming-request-data-in-flask
  - "Leveraging Large Language Models to Improve REST API Testing", https://arxiv.org/pdf/2312.00894.pdf Uses LLMs to extract parameter types, constraints on parameters, etc. Is able to generate much more accurate REST test values.
  - "RESTTESTGEN: Automated Black-Box Testing of RESTful APIs", https://profs.scienze.univr.it/~ceccato/papers/2020/icst2020api.pdf
  - "COTS: Connected OpenAPI Test Synthesis for RESTful Applications", https://arxiv.org/pdf/2404.19614
  - M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated test generation for REST APIs: No time to rest yet," in Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 289–301, 2022.

References (9)

- REST and OpenAPI API testing tools:
  - “Leveraging Large Language Models to Improve REST API Testing”, https://arxiv.org/pdf/2312.00894.pdf Uses LLMs to extract parameter types, constraints on parameters, etc.   Is able to generate much more accurate REST test values.
  - “RESTTESTGEN: Automated Black-Box Testing of RESTful APIs”, https://profs.scienze.univr.it/~ceccato/papers/2020/icst2020api.pdf
  - “COTS: Connected OpenAPI Test Synthesis for RESTful Applications”, https://arxiv.org/pdf/2404.19614
  - M. Kim, Q. Xin, S. Sinha, and A. Orso, “Automated test generation for REST APIs: No time to rest yet,” in Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 289–301, 2022.
  - “APITestGenie: Automated API Test Generation through Generative AI ”, https://arxiv.org/pdf/2409.03838
  - “Preprint: DeepREST: Automated Test Case Generation for REST APIs Exploiting Deep Reinforcement Learning”, https://arxiv.org/pdf/2408.08594
  - ”A Multi-Agent Approach for REST API Testing with Semantic Graphs and LLM-Driven Inputs”, https://arxiv.org/pdf/2411.07098

References (10)

- REST and OpenAPI API:
  - “A Public Benchmark of REST APIs”, Alix Decrop et. al., Proceedings - 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories, MSR 2025

References (11)

- API alternatives to REST:
  - OpenAPI: https://swagger.io/docs/specification/basic-structure/
  - OpenAPI: https://support.smartbear.com/swaggerhub/docs/en/get-started/openapi-3-0-tutorial.html
  - gRPC: https://grpc.io/
- Comparing alternatives:
  - https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/
  - https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them
  - https://medium.com/@anthonydmays/grpc-vs-rest-comparing-api-styles-in-practice-28d2a7c9a349
  - https://blog.bitsrc.io/rest-vs-graphql-vs-grpc-684edfacf810
  - https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da
  - https://rapidapi.com/guides/difference-kafka-rest
  - https://medium.com/@vikalprusia/kafka-vs-http-757c6d49e23
  - https://blog.postman.com/graphql-vs-rest