

.NET Advanced

REST & RESTful API

Hoe moet ik mijn web API nu juist ontwerpen ?

<http://www.example.com/api/v1/People/3>

En waarom bijvoorbeeld niet :

<http://www.example.com/api/v1/Person/3>

of :

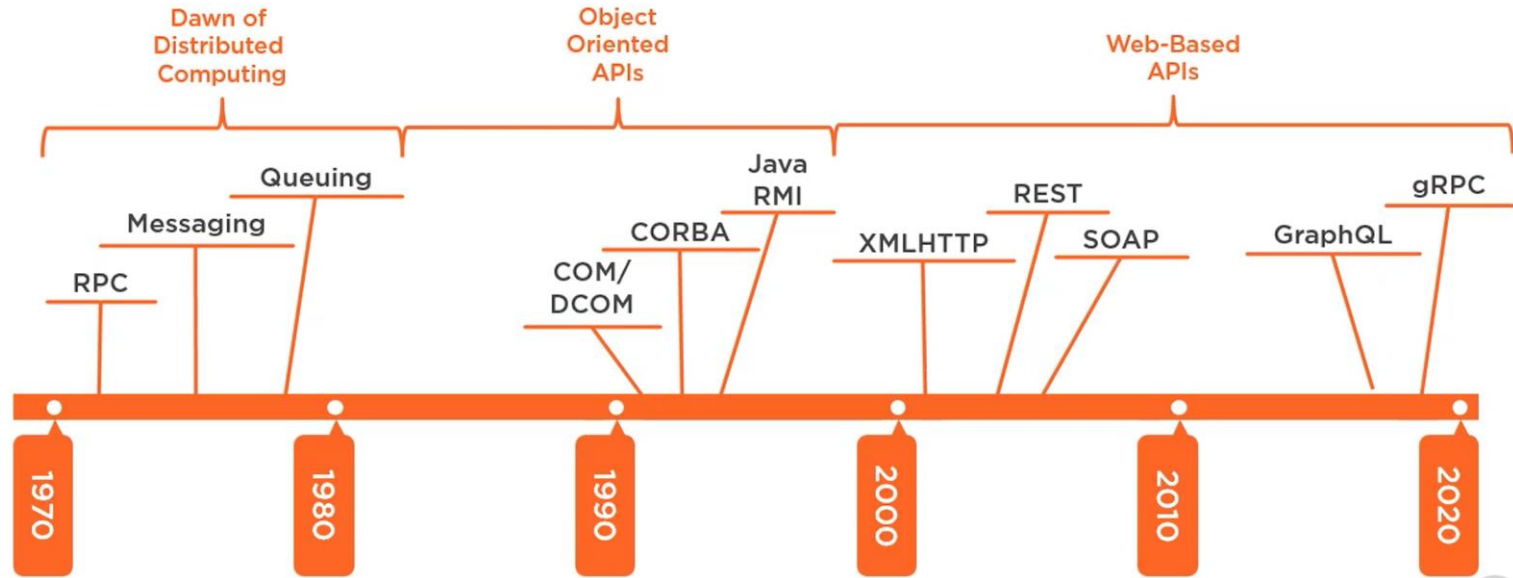
<http://www.example.com/api/v1/Person?id=3>

of :

<http://www.example.com/GetPerson?id=3>

Geschiedenis...

The History of Distributed APIs



REST API

- REST staat voor ‘Representational State Transfer’
- Alhoewel het geen verplichting is maakt de meerderheid van de bestaande REST API uiteraard gebruik van het **HTTP protocol**.
- Alhoewel het gebruik van “user-friendly” URL’s deel uitmaakt van REST is dat zeker **niet het enige** dat nodig is om een goed REST-ful ontwerp te maken.

Uncleaned URL	Clean URL
<code>http://example.com/index.php?page=name</code>	<code>http://example.com/name</code>
<code>http://example.com/about.html</code>	<code>http://example.com/about</code>
<code>http://example.com/index.php?page=consulting/marketing</code>	<code>http://example.com/consulting/marketing</code>
<code>http://example.com/products?category=12&pid=25</code>	<code>http://example.com/products/12/25</code>

- REST is geen protocol op zich, het is een soort van “style guidelines” voor het ontwerpen van services

History [\[edit \]](#)

[Roy Fielding](#) defined REST in his 2000 PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures" at [UC Irvine](#).^[2] He developed the REST architectural style in parallel with [HTTP 1.1](#) of 1996–1999, based on the existing design of [HTTP 1.0](#)^[7] of 1996.

In a retrospective look at the development of REST, Fielding said:

Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do within a process that accepts proposals from anyone on a topic that was rapidly becoming the center of an entire industry. I had comments from well over 500 developers, many of whom were distinguished engineers with decades of experience, and I had to explain everything from the most abstract notions of Web interaction to the finest details of HTTP syntax. That process honed my model down to a core set of principles, properties, and constraints that are now called REST.^[7]



HTTP protocol als transportmiddel

- Vermits onze REST API via het http protocol zal werken moeten we ons daar op richten om te bekijken wat de mogelijkheden zijn. In het HTTP protocol zit een HTTP **request** en een HTTP **response** en kunnen we deze zaken hierin telkens onderscheiden:

- HTTP Request :

- **URL**

- In de **URL** zit :

- **Protocol of scheme** (http of https)
 - **Domeinnaam** van de server
 - **Poort** (of standaard - voor http- port 80)
 - **Path:** dit is een vrij gedeelte dat we kunnen gebruiken voor onze API
 - dit is de **Route** die we kunnen instellen op niveau van controller en/of action
 - **Query parameters:** dit is een lijst van “name & value pairs”
 - Deze zullen automatisch worden doorgegeven aan de methode (action)

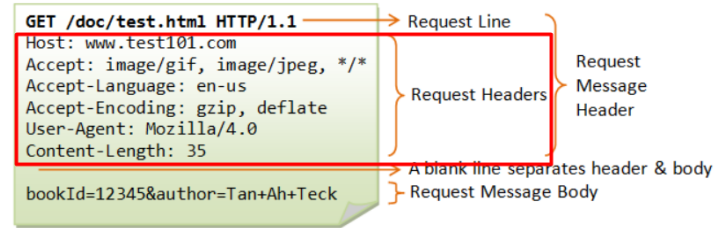


```
[Route("api/People")]
public ActionResult<IEnumerable<Person>> GetPeople(string firstName, string lastname,
string sortBy, int pageNr, string dir = "asc", int pageSize = 10)
{
```

HTTP protocol als transportmiddel

- HTTP Request:

- **Headers:** Ook de headers die in de request aanwezig zijn kunnen voor de API worden gebruikt. Dit zijn eveneens “name & value pairs”. (zie ook verder)



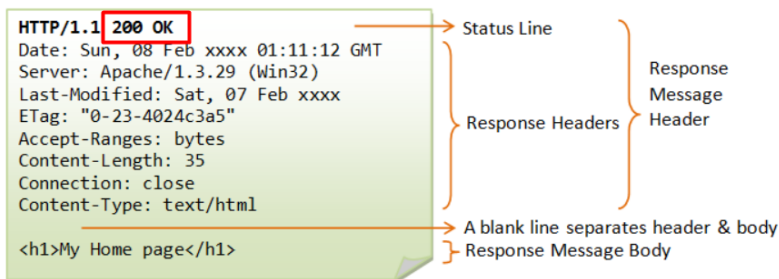
- **Body:**
 - Ook de body van de request kan (JSON) data bevatten, bij een GET niet, maar wel bij een POST, PUT of PATCH.
- Het ASP framework zal trachten zelf op zoek te gaan naar de nodige parameters, maar je kan dit ook zelf sturen door dit aan te geven met de nodige attributen.

```
[Route("api/People")]
0 references
public ActionResult<IEnumerable<Person>> GetPeople([FromQuery] string firstName, [FromHeader] string lastname,
{
    [FromBody] string sortBy, int pageNr, string dir = "asc", int pageSize = 10)
{
```

HTTP protocol als transportmiddel

- HTTP response:

- **Status code:** de status code kunnen we sturen vanuit onze API, het ASP framework voorziet hiervoor ook een aantal helper methodes op de controller base klasse.



```
[HttpPost]
0 references
public IActionResult AddPerson([FromBody]Person p)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    ctxt.People.Add(p);
    ctxt.SaveChanges();

    return Created("", p);
}
```

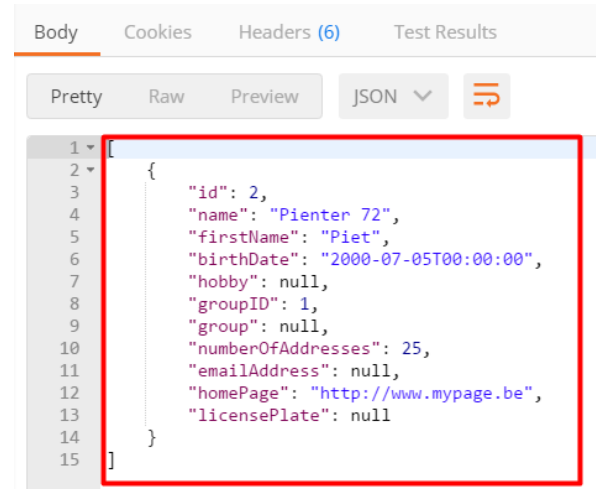
- **Headers:** ook deze name & value pairs kunnen worden ingesteld vanuit onze API om op die manier extra info te kunnen meegeven over de geleverde response

```
result = result.Skip(pageNr * pageSize);
result = result.Take(pageSize);

Response.Headers.Add("X-Total-Count", result.Count().ToString());
```


HTTP protocol als transportmiddel

- HTTP Response:
 - **Body:** De body zal bij de response uiteraard de gevraagde gegevens bevatten. Het formaat (veelal JSON) kan worden aangegeven met een header veld (Content-type) en kan vanuit de request worden gestuurd (bv. De client heeft liefst XML of JSON, ..)



```
1 [
2   {
3     "id": 2,
4     "name": "Pienter 72",
5     "firstName": "Piet",
6     "birthDate": "2000-07-05T00:00:00",
7     "hobby": null,
8     "groupID": 1,
9     "group": null,
10    "numberOfAddresses": 25,
11    "emailAddress": null,
12    "homePage": "http://www.mypage.be",
13    "licensePlate": null
14  }
15 ]
```


Samenstelling van de URL



- Aan de hand van het “scheme” en de “host name” zal de request (via het internet) gestuurd worden naar de juiste (web) server.
- Eens aangekomen op de server zal de web server zelf de verdere afhandeling op zich nemen. Dit aan de hand van bepaalde routeringsinstellingen (zoals bv. de route van een ASP Controller).
- Het is best mogelijk (en zelfs veelvoorkomend) dat binnen eenzelfde domein (server) zowel een website (of webapplicatie) ALS een web API wordt gehuisvest. Er moet dan een eerste opsplitsing worden genomen aan de hand van de URL.
- Veelal gebeurt dit door het **path** te laten starten met de string ‘api’:
 - <http://www.myserver.com/index.html> => URL naar de website
 - <http://www.myserver.com/api/people> => URL naar de API
- Alternatief is dat men een ander **subdomein** gebruikt
 - <http://www.myserver.com/index.html> => URL naar de website
 - <http://api.myserver.com/people> => URL naar de API (subdomein)
- **Nadeel** bij het gebruik van subdomein is dat men rekening moet houden met **CORS** problematiek ! (zie verder)

Versioning

- Eens een API online staat en er (meerdere) clients gebruik van maken waarop je zelf geen vat hebt dan kan je de API niet meer zomaar **fundamenteel** wijzigen.
- Indien je aanpassingen moet doen die niet meer “backwards compatible” zijn is het beter om de bestaande API in dienst te laten en een 2^e bijkomende te voorzien.
- Een veelgebruikte manier is om de versie in de **URL** te verwerken:
 - <http://www.myserver.com/api/people> (de eerste versie)
 - <http://www.myserver.com/api/v2/people> (een 2e versie van de API)
 - de bestaande clients kunnen eventueel blijven werken met de oude versie.
- Soms wordt de versie ook via de **Query Parameters** doorgegeven
 - <http://www.myserver.com/api/people?version=2>
- Of via een **header** veld:

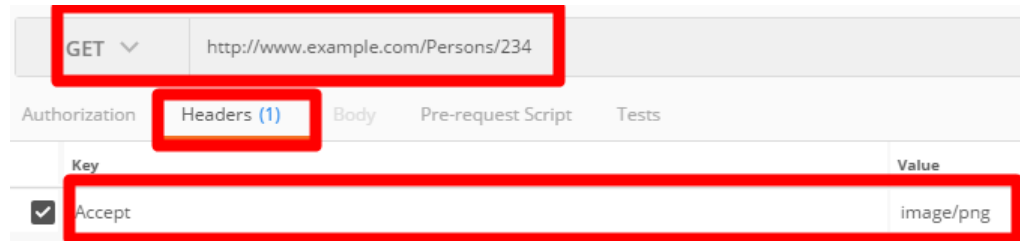
GET ▾		http://www.myserver.com/api/people			
Authorization		Headers (2)	Body	Pre-request Script	Tests
	Key				Value
<input checked="" type="checkbox"/>	Accept				application/json
<input checked="" type="checkbox"/>	X-Version				2.0
	New key				Value

REST API design principles (1)

- REST APIs are designed around **resources**, which are any kind of object, data, or service that can be accessed by the client. Typically: a noun (zelfstandig naamwoord) is used
 - <http://www.example.com/customers>
 - <http://www.example.com/products>
 - <http://www.example.com/orders>
 - ...
 - <http://adventure-works.com/create-order> // **Dus niet dit ! Geen werkwoorden**
- Adopt a **consistent naming convention** in URIs. In general, it helps to **use plural nouns** for URIs that reference collections.
- A resource has an **identifier**, which is a URI that uniquely identifies that resource. For example, the URI for a particular customer order might be:
 - <http://adventure-works.com/orders/1>
 - <http://www.example.com/customers/33245>
 - <http://www.example.com/products/66432>
 - ...

REST API Design principles (2)

- Clients interact with a service by exchanging **representations of resources**. Many web APIs use **JSON** as the exchange format. For example, a GET request to the URI listed above might return this response body:
 - {"orderId":1,"orderValue":99.90,"productId":1,"quantity":1}
- Media types
 - In the HTTP protocol, formats are specified through the use of *media types*, also called **MIME types**. For non-binary data, most web APIs support **JSON** (media type = application/json) and possibly **XML** (media type = application/xml).
 - The Content-Type header in a request or response specifies the format of the representation.



REST API Design principles (3)

- REST APIs use a uniform interface, which helps to decouple the client and service implementations. For REST APIs built on HTTP, the uniform interface includes using standard **HTTP verbs** perform operations on resources. The most common operations are GET, POST, PUT, PATCH, and DELETE.

~~http://www.example.com/GetPerson?id=3~~

- **GET** retrieves a representation of the resource at the specified URI. The body of the response message contains the details of the requested resource.
- **POST** creates a new resource at the specified URI. The body of the request message provides the details of the new resource. Note that POST can also be used to trigger operations that don't actually create resources.
- **PUT** either creates or replaces the resource at the specified URI. The body of the request message specifies the resource to be created or updated.
- **PATCH** performs a partial update of a resource. The request body specifies the set of changes to apply to the resource.
- **DELETE** removes the resource at the specified URI.

PUT versus PATCH

PUT http://localhost:3000/contacts/1

Authorization Headers (1) Body Pre-req

form-data x-www-form-urlencoded raw

```
1 {
2   "first_name": "Jason",
3   "last_name": "Arnold",
4   "location": "NYC",
5   "email": "jason.arnold@test.com"
6 }
7
```

A PUT request to /contacts/1

```
{
  "contacts": [
    {
      "id": 1,
      "firstName": "Jason",
      "lastName": "Arnold",
      "location": "NYC",
      "email": "jason@test.com"
    },
  ],
}
```

Originele data

PATCH http://localhost:3000/contacts/1

Authorization Headers (1) Body Pre-request Script

form-data x-www-form-urlencoded raw binary JS

```
1 {
2   "location": "Brooklyn"
3 }
4
```

```
{
  "contacts": [
    {
      "first_name": "Jason",
      "last_name": "Arnold",
      "location": "Brooklyn",
      "email": "jason.arnold@test.com",
      "id": 1
    },
  ],
}
```

REST API Design principles (4)

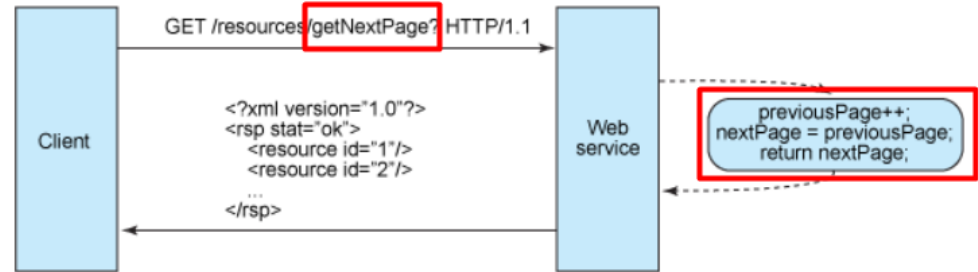
- REST APIs use a **stateless request model**.
 - HTTP requests should be **independent and may occur in any order**, so keeping transient state information between requests is not feasible.
 - The only place where information is stored is in the resources themselves, and **each request should be an atomic operation (= een op zichzelf staande operatie)**.
 - This constraint enables web services to be highly **scalable**, because there is no need to retain any affinity between clients and specific servers.
 - **Any server** can handle **any request** from **any client**.

“It is critical to build a scalable architecture in order to take advantage of a scalable infrastructure”

Stateful versus Stateless server

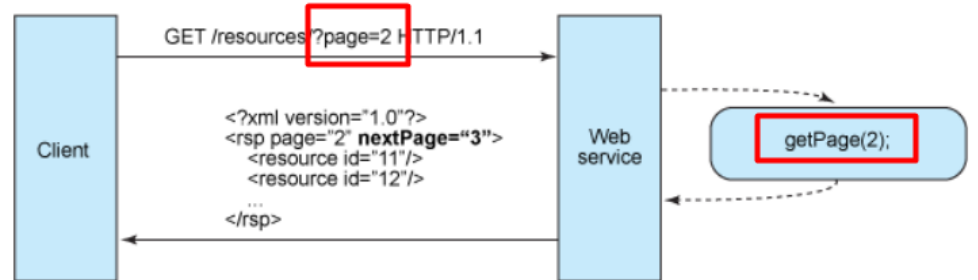
- **Stateful :**

- De server houdt “state” bij
- bv. wat is de huidige pagina.
- De client vraagt telkens: geef me de volgende of vorige pagina



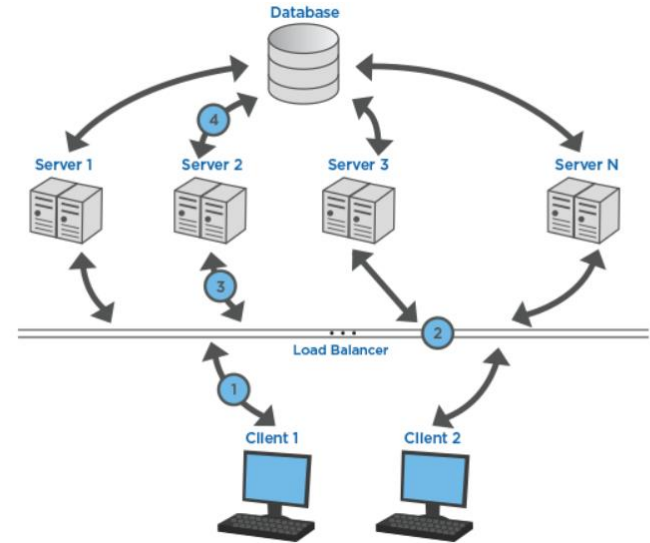
- **Stateless:**

- De server houdt geen enkele state bij.
- Elke request moet de nodige info bevatten om deze te kunnen beantwoorden.
- Dus de page nr, komt mee in elke request.
- De client houdt de state (bv. huidige pagina) bij.



Wat is schaalbaarheid

- Bij een “scalable” systeem kan er met meerdere servers gewerkt worden.
- Er worden automatisch meer servers opgestart (= scale out) wanneer het aantal requests / min. Verhoogd => “cloud” of “server farm”
- Tussen clients en servers bevindt zich een “load balancer”.
- De “load balancer” zal de inkomende requests verdelen naar de servers, bv:
 - Request 1 van client 1 naar server 1
 - Request 1 van client 2 naar server 2
 - Request 2 van client 1 naar server 3
 - ...
- **Maw. requests van eenzelfde client worden mogelijk afgehandeld door verschillende servers**



REST API Design principles (5)

- **Hypermedia As The Engine Of Application State (HATEOAS)**
- REST APIs are driven by **hypermedia links** that are contained in the representation.
- Het idee is dat elke opgevraagde resource “zelf beschrijvend” wordt.
- In dit voorbeeld wordt bij het opvragen van een persoon de **url** meegegeven van de resource zelf (**persoon**), en ook van zijn **groep**
- Als de client meer info wenst te bekomen over deze groep kan deze gebruik gaan maken van deze URL om een GET request te doen.
- Indien er geen hypermedia link aanwezig zou zijn:
 - Dan moet de client zelf de link gaan samenstellen met behulp van de group Id.

```
{
  "id": 1,
  "name": "Pienter 1",
  "firstName": "Piet",
  "group": {
    "id": 1,
    "name": "2EA1",
    "_links": {
      "self": "https://localhost:44388/api/groups/1"
    }
  },
  "_links": {
    "self": "https://localhost:44388/api/people/1"
  }
}
```

Hypermedia

- Met hypermedia kan je heel ver gaan...
- In dit voorbeeld wordt bij elke resource (TODO items) ook de hypermedia links aangereikt om het item te verwijderen of aan te passen.
- Idee is dat er zo weinig mogelijk “hardcoded” zaken met betrekking tot de URL's moet worden ingebakken in de client(s) van de API.
- De API reikt zelf de nodige URL's aan.
- Op die manier is de client minder onderhevig aan wijzigingen wat betreft de URL's en kan de maker van de API op eender welk moment zijn API aanpassen zonder dat er aanpassingen nodig zijn in de client code.

GET <http://localhost:5000/api/todo/1>

Authorization Headers (1) Body Pre-request Script Tests

Key	Value
<input checked="" type="checkbox"/> Accept	application/json+hateoas
New key	Value

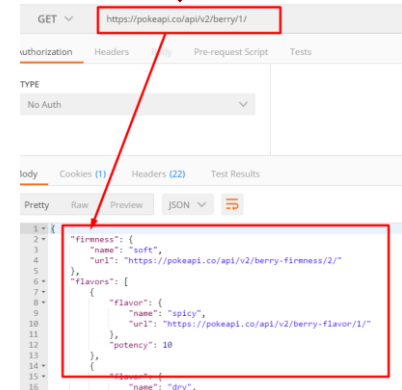
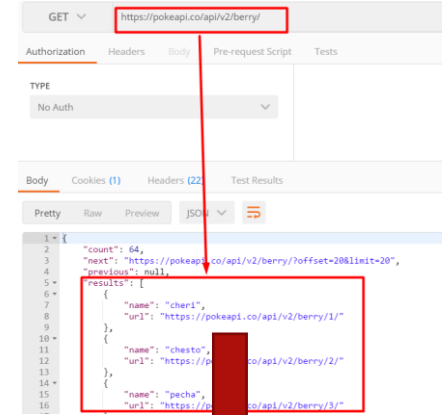
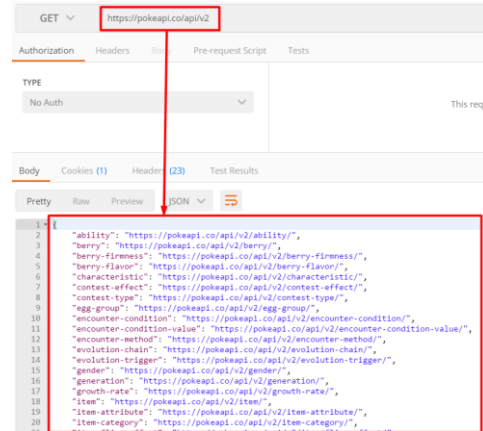
Body Cookies Headers (4) Test Results

Pretty Raw Preview JSON

```
1 {
2   "value": {
3     "id": 1,
4     "name": "Item1",
5     "isComplete": false
6   },
7   "links": [
8     {
9       "href": "http://localhost:5000/api/ToDo/1",
10      "rel": "self",
11      "method": "GET"
12    },
13    {
14      "href": "http://localhost:5000/api/ToDo/1",
15      "rel": "put-todo",
16      "method": "PUT"
17    },
18    {
19      "href": "http://localhost:5000/api/ToDo/1",
20      "rel": "delete-todo",
21      "method": "DELETE"
22    }
23  ]
24 }
```

Hypermedia

- Soms geeft een API via de root URL een overzicht van alle beschikbare resources in deze API
- De API wordt op deze manier “zelf beschrijvend”, alhoewel er niet echt een richtlijn is om aan te geven hoe deze URL's best moeten verwerkt worden in de data.
- Je zal daarom zien dat het niet in elke publieke API op dezelfde manier gebeurt.

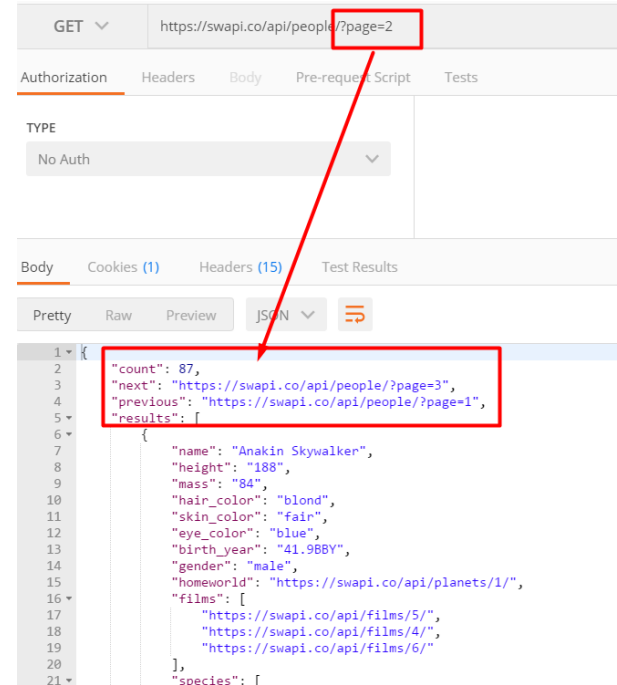


REST API filtering, paging, sorting,...

- Gebruik query string om te filteren op andere attributen:
 - <http://www.example.com/customers?firstName=john>
 - <http://www.example.com/customers?firstName=john&age=23>
- Voorzie ook een manier om de hoeveelheid gegevens te beperken om onnodige datatransfer te voorkomen (bv. paging):
 - <http://www.example.com/orders?limit=25&offset=50>
 - Gebruik een “default” wanneer de client deze niet opgeeft via de URL (bv. limit = 10)
- Voorzie een manier om alle attributen niet onnodig over te moeten halen:
 - <http://www.example.com/orders?fields=ProductID,Quantity>
- Gelijkaardig zou je ook sortering kunnen voorzien:
 - <http://www.example.com/people?sort=LastName,firstName>
 - <http://www.example.com/people?sort=age:asc,firstName:desc>

Paging

- Paging : opvragen van een bepaalde “page” wordt bij sommige API’s ook wel via de header velden gedaan ipv. via de URL
- In de response zit er dikwijls ook meer info omtrent de pages, bv:
 - Aantal resultaten
 - Link naar volgende/vorige/eerste/laatste pagina’s
 - Soms zit deze info niet bij in de **body** maar in de **header** velden van de response



REST API resource to entity

- A resource does not have to be based on a single physical data item.
- For example, an order resource might be implemented internally as several tables in a relational database, but presented to the client as a single entity.
- Avoid creating APIs that simply mirror the internal structure of a database.
- The purpose of REST is to model entities and the operations that an application can perform on those entities.
- A client should not be exposed to the internal implementation.
- Also consider the relationships between different types of resources and how you might expose these associations.
 - <http://www.example.com/customers/5/orders> (get all of the orders for customer 5)
 - <http://www.example.com/orders/99/customer> (get the customer for that specific order)
 - Avoid requiring resource URIs more complex than *collection/item/collection*.
 - Alternative: HATEOAS

REST API HTTP verbs

- Some examples:

Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

HTTP Status codes

- Use the status codes to indicate the result of the request
- Additionally include extra info about the error in the body of the response



HTTP Response codes

CODE	Constant	Description
200	OK	Successfully executed
201	CREATED	Successfully executed and a new resources has been created. The response body is either empty or contains a URI(s) of the created resource. The location header should also contain the new URI
202	ACCEPTED	The request was valid and has been accepted but has not yet been processed. The response should include a URI to poll for status updates on the request. Allows for asynchronous request
204	NO_CONTENT	The request was successful but the server did not have a response
301	MOVED_PERMANENTLY	The new location should be returned in the response
400	BAD_REQUEST	Malformed or invalid request
401	UNAUTHORIZED	Invalid authorization credentials
403	FORBIDDEN	Disallowed request. Security error
404	NOT_FOUND	Resource not found
405	METHOD_NOT_ALLOWED	Method (verb) not allowed for requested resource. Response will provide an "Allow" header to indicate what is allowed
415	UNSUPPORTED_MEDIA_TYPE	Client submitted a media type that is incompatible for the specified resource.
500	INTERNAL_SERVER_ERROR	Catchall for server processing problem
503	SERVICE_UNAVAILABLE	Response in the face of too many request

References

- <https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design>
- <https://specs.openstack.org/openstack/api-wg/index.html>
- <https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md>
- <http://restcookbook.com/>
- <https://mathieu.fenniak.net/the-api-checklist/>
- <http://www.restapitutorial.com/httpstatuscodes.html>
- Overzicht van APIs op het internet:
- http://www.programmableweb.com/category/all/apis?order=field_popularity

Onderzoek

- Hoe RESTFul zijn deze API ?
- <https://www.pokeapi.co/>
- <https://swapi.co/>
- <https://anapiofficeandfire.com/>
- <https://api.openaq.org>
(<https://docs.openaq.org/>)
- welke resources kan je opvragen ?
- welke representaties worden ondersteunt ?
- wat is de default representatie ?
- welke pagingmogelijkheden
- welke opzoek/filtermogelijkheden,..
- In hoeverre is er hypermedia aanwezig ?
- welke verbs worden ondersteund ?
- hoe navigeer ik naar relationships ?
- Welke status codes worden gebruikt ?
- Hoe gaat men om met verschillende versies van interface ?
- Andere features ?

Onderzoek API:

- Wat kan ik met de API doen ?
 - Welk protocol gebruik ik ?
 - Wat kan ik opvragen ?
 - Hoe vraag ik iets op ?
 - Wat krijg ik terug ?
 - Hoe bepaal ik wat ik wil terugkrijgen ?
 - Hoe kan ik zoeken ?
 - Hoe / kan ik iets aanpassen ?
 - Hoe kan ik navigeren naar gerelateerde data ?
 - Is er een beveiliging ?
 - Moet ik me aanmelden ?
 - Hoe kom ik te weten welke properties ik allemaal kan verwachten in de gegevens ?
 - Hoe weet ik dat er iets mis is met mijn request ? En wat er foutgelopen is ?
- Hoe gaan we om met grote hoeveelheden gegevens ?

