Service-Oriented Architectures and API Design

(borrowed and edited from https://dzone.com/articles/characteristics-of-good-apis-daedtech)

1. Practice First: A simple HTTP service on Heroku

Before discussing APIs and REST services, lets build and deploy a simple service.

1.1. Server

Create a project on github or use an existing one. If you start from github, set the .gitignore file for nodejs project as it will make your life easier (go read the content of gitignore)

Clone it locally, cd to it

Create a file called `Procfile (called exactly like this)`

In it write this:

```
web: node api.js
```

(instead of app.js you can put the name of the file where you put your main server code)

Run

```
npm init
```

```
npm install express body-parser --save
```

Create a file called api.js with this content:

```
const express = require('express')
const app = express()
const PORT = process.env.PORT || 3000

var courses_offered = [{id: 21, name: 'HCI'},
{id: 28, name:'sweng'}]
```

```
app.get('/', (req, res) => res.send('Hello World!'))

app.get('/courses', (req, res) => {
   res.json(courses_offered)
})

app.listen(PORT, () => console.log('Example app listening on port'+
PORT))
```

Make sure now that your package.json file contains the **start** entry pointing to whatever is the main file of your application:

```
  "scripts": {
    "start": "node api.js",
<you may have something else here>
  },
```

Now let's test it out: run your app:

```
npm start
```

And access http://localhost:3000 to see that you get actually the reply hello world

If all works well, kill the app, and add/commit/push your code (remember to add the Procfile and package.json to your commit, but NOT node_modules)

1.2. Client

Create a file client.js (or call it as you like, and put it where you like)

```
const fetch = require("node-fetch");
const url = "https://api.sunrise-sunset.org/json?lat=36.7201600&lng=-4.4203400"

const getLocation = async url => {

  try {
    const response = await fetch(url);
    const json = await response.json();
    console.log(json)
  } catch (error) {
    console.log(error);
  }
};


getLocation(url);
```

You can run it and see what happens, or you can change the url to point to localhost:3000/courses after you started up the server

1.3. Deployment on Heroku

Create an account at heroku.com

Create a new app

In **deployment methods**, choose connect to github

In the **connect to github** section, click the connect button

Then select the repo you created on github and connect to that

For now we accept the defaults: we do NOT enable automatic deploys, and we deploy off master.

Now, click deploy! And voila', the app is live.

Congratulations for creating your first API and deploying in on Heroku

1.4. Documentation (API definition)

There are many ways to document an API, in this course we use openAPI 2.0, and apiary as the editor

- Create an account at apiary.io
- Create a new API, link it to your github repo (so that the api definition document is under version control)
- Modify the api, for example to get to something like this (use your heroku app url)

```yaml
swagger: '2.0'
info:
  version: '1.0'
  title: "Test api"
  description: Get info on courses.
  license:
    name: MIT
    url: https://github.com/apiaryio/polls-api/blob/master/LICENSE
host: ts2myfirstapi.herokuapp.com
basePath: /
schemes:
- https
consumes:
- application/json
produces:
- application/json
paths:
  /courses:
    x-summary: Course management
    get:
      summary: List All Courses
      responses:
        200:
          description: Successful Response
          schema:
            type: array
            items:
              $ref: '#/definitions/Course'
          examples:
            application/json:
              - id: 11
                name: 'c1'
              - id: 33
                name: 'some other course'

    post:
      description: Create a course
      summary: Create a course
      parameters:
        - name: course_name
          in: body
          required: true
          schema:
            type: integer
      responses:
        201:
          description: ''
          schema:
            $ref: '#/definitions/Course'
          examples:
            application/json:
                id: 38
                name: 'some course'

definitions:
  Course:
    title: some titme
    type: object
    properties:
      id:
        type: integer
      name:
        type: string
    required:
      - id
      - name
```

This is only a super basic API, but it's just to get started with heroku and api description languages

2. Application Programming Interfaces and Service-Oriented Architectures

2.1. APIs

An Application Programming Interface (API) defines how other programs can interact with your software. There are millions of examples (just google API example), but you can for example look at these ones:

- Instagram API: https://developers.facebook.com/docs/instagram-api
- Github: https://developer.github.com/v3/
- UK Police: https://data.police.uk/docs/
- Sunset and sunrise: https://sunrise-sunset.org/api
- Air quality data: https://docs.openaq.org/

DISCUSS: why do these companies expose APIs? What could we do if all unitn/esse3/ other trentino services had a nicely exposed APIs?

DISCUSS: Take a look at these API and let's discuss them. Notice also how they are documented, auth issues, rate limits.

Web developers think of APIs as HTTP endpoints. In contrast, desktop developers may think of APIs as "library code" written by developers from other organizations. Folks writing low-level code like drivers? An API provides the hooks into OS system calls.

Thus, each of these personae has the right answer, but a small slice of it. As you write code, ask yourself about the user of that code. **If that user interacts with your code via code of their own, you're building an API**. And before you ask, yes, **that even applies to other developers in your company or even your team that use your code.**

Consequently, **understanding properties of good APIs carries vital importance for our industry. It makes the difference between others building on your work or avoiding it**. And, let's be honest — there's a certain amount of professional pride at stake. So what makes APIs good? Let's consider some characteristics of good APIs.

Simplicity

Programmers tend to solve complex problems, making it easy to let that complexity bleed through to users of our APIs. **Keeping them simple requires work, and sometimes presents a serious challenge.**

Our tendency to want to be helpful adds to the difficulty. Often, programmers want to offer several different ways to do something. "Maybe they want to pass these dependencies in through a constructor. But what if they prefer to use setters? We should make both available." The road to complexity can be paved with good intentions.

Fight the urge to add undue complexity, even when you think it might help. Good APIs offer simplicity, and it takes a good bit of effort to preserve that simplicity.

Provide Useful Abstractions

Next, consider the concept of abstraction. When you successfully hide details from users of your API, leaving only the essentials, you offer them abstraction.

Examples of abstraction abound in our world. Device drivers abstract the details of dealing with vendor hardware. Threading models provide an abstraction for worrying about scheduling execution at the OS level. The OS itself provides an abstraction over the differences in core computer hardware. Even your programming language abstracts away the details of writing machine code.

Well written code provides abstractions, and APIs are no exception. Your API should hide the details of what it does from your users while making itself useful to them. If your users need to dive into your code or execution to understand, you have provided a poor abstraction.

Consistent

Examine your API for **consistency**. Consistency is relatively easy to understand. Don't call them "users" sometimes and "customers" at other times. Name the same things the same way. Have a common style.

If you have an API for file access that lets you call Open(string filename), you should also offer a Close(string filename) method. Open and close have opposite meanings, so offering both creates symmetry as concerns that operation. Calling the Close method "Destroy" or, worse, not having a close method, would create a great deal of confusion.

Follow the Principle of Least Astonishment

For the last consideration in the mix, I want to introduce you to the cleverly named "principle of least astonishment." This holds that "a system should behave in a manner consistent with how users of that component are likely to expect it to behave; that is, users should not be astonished at the way it behaves." To put it more bluntly, **don't write APIs that flabbergast your users**. **NO SURPRISES. DON'T TRY TO BE TOO SMART.**

Edited from wikipedia: A textbook formulation is: "People are part of the system. The design should match the user's experience, expectations, and mental models."The choice of "least surprising" behavior can depend on the expected audience (for example, end users, programmers, or system administrators). In more practical terms, **the principle aims to leverage the pre-existing knowledge of users to minimize the learning curve**, for instance by designing interfaces that borrow heavily from "functionally similar or analogous programs with which your users are likely to be familiar". User expectations in this respect may be closely related to a particular computing platform or tradition. This practice also involves the application of sensible defaults. When two elements of an interface conflict, or are ambiguous, the behavior should be that which will least surprise the user; in particular a programmer should try to think of the behavior that will least surprise someone who uses the program, rather than that behavior that is natural from knowing the inner workings of the program. In programming, a good example of this principle is the common ParseInteger(string, radix) function which exists in most languages and is used to convert a string to an integer value. The radix is usually an optional argument and assumed to be 10 (representing base 10). Other bases are usually supported (like binary or octal) but only when specified explicitly; when the radix argument is not specified, base 10 is assumed. Notably JavaScript did not originally adopt this behavior, which resulted in developer confusion and software bugs.

Think of Your API as a Product

Simple, useful, consistent, and predictable all describe not only good APIs but good products. That's no accident. When you write APIs, you create a product. Whether you think of it that way or not and whether you actually sell it or not, you have a product on your hands, intended for use by someone else.

As developers, we can easily lose sight of that. **Subject to the curse of knowledge, we assume that our API users, fellow programmers, will know what we know and figure out what we've figured out**. **We don't naturally think of them as our end-users or our customers**.

Overcoming that tendency and putting ourselves in their shoes is the unifying theme behind good API design. So when you write your next API, put yourself in the shoes of your customer, and imagine what you would want.

2.2. SOA

Thinking in terms of APIs helps us think in terms of exposing our functionality as a **service** for other people to consume. Services are essentially programs that can be accessed through API, typically via http (hence the commonly used term *web services*). A good way to start discussing services is to remember the famous jeff bezos' email to all employees, almost 18 years ago, which goes like this:

1. *All teams will henceforth expose their data and functionality through service interfaces.*
2. *Teams must communicate with each other through these interfaces.*
3. *There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.*
4. *It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols – doesn't matter. Bezos doesn't care.*
5. *All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.*
6. *Anyone who doesn't do this will be fired.*

The bottom line is that we must think of building our code in terms of functions we offer to other services, and make sure to document these APIs well enough so that other people can understand how to use them correctly. And this also means they should follow the properties of good interfaces discussed above. When we do this, we have modularized the code, made it easy to test and use, and documented every data exchange point and format. Any change to a service interface will have to be properly managed - it will have to be communicated, it must be thought how to address backward compatibility, and so on. You tend not to think much about this things if the interaction simply happens through some shared data store or some other mean.

A service-oriented architecture is a software architecture where you organize your modules in terms of services that operate independently and interact with other services via APIs.

3. Introduction to REST

3.1. REST basics

(largely borrowed from
https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design)

In 2000, Roy Fielding proposed Representational State Transfer (REST) as an architectural approach to designing web services. REST is an architectural style for building distributed systems based on hypermedia. REST is independent of any underlying protocol and is not necessarily tied to HTTP. However, most common REST implementations use HTTP as the application protocol, and this guide focuses on designing REST APIs for HTTP.

Here are some of the main design principles of RESTful APIs using HTTP: **REST APIs are designed around resources, which are any kind of object that can be accessed by the client**. **In a first, high level approximation, think of them as the main entities in your application. For esse3, they might be students, courses, teachers, ...**

A resource (e.g., a course) has an identifier, which is a URI that uniquely identifies that resource. For example, the URI for a particular customer order might be:

http://universityexample.com/courses/15734

Clients interact with a service by exchanging **representations** of resources. Many web APIs use JSON as the exchange format, but there is noting in REST that says that json must be used.

For REST APIs built on HTTP, the uniform interface includes using standard HTTP verbs to perform operations on resources. The most common operations are GET, POST, PUT, PATCH, and DELETE.

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. 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**. That said, other factors can limit scalability.

3.2. Organize the API around resources

Focus on the business entities that the web API exposes. For example, in an e-commerce system, the primary entities might be customers and orders. Creating an order can be achieved by sending an HTTP POST request that contains the order information. The HTTP response indicates whether the order was placed successfully or not. **When possible, resource URIs should be based on nouns (the resource) and not verbs (the operations on the resource)**.

HTTP

http://adventure-works.com/orders   === **Good**

http://adventure-works.com/create-order === **Avoid!!**

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.

Entities are often grouped together into collections (e.g, orders, or customers). A collection is a separate resource from the item within the collection, and should have its own URI. For example, the following URI might represent the collection of orders:

http://adventure-works.com/orders

Sending an HTTP GET request to the collection URI retrieves a list of items in the collection. Each item in the collection also has its own unique URI. An HTTP GET request to the item's URI returns the details of that item.

Adopt a consistent naming convention in URIs. In general, it helps to use plural nouns for URIs that reference collections. **It's a good practice to organize URIs for collections and items into a hierarchy. For example, /customers is the path to the customers collection, and /customers/5 is the path to the customer with ID equal to 5**. This approach helps to keep the web API intuitive. Also, many web API frameworks can route requests based on parameterized URI paths, so you could define a route for the path /customers/{id}.

Also consider the relationships between different types of resources and how you might expose these associations. For example, the /customers/5/orders might represent all of the

orders for customer 5. You could also go in the other direction, and represent the association from an order back to a customer with a URI such as /orders/99/customer. However, extending this model too far can become cumbersome to implement. Another solution is to provide navigable links to associated resources in the body of the HTTP response message. This mechanism is described in more detail in the section Using the HATEOAS Approach to Enable Navigation To Related Resources later.

In more complex systems, it can be tempting to provide URIs that enable a client to navigate through several levels of relationships, such as /customers/1/orders/99/products. However, this level of complexity can be difficult to maintain and is inflexible if the relationships between resources change in the future. Instead, **try to keep URIs relatively simple**. Once an application has a reference to a resource, it should be possible to use this reference to find items related to that resource. The preceding query can be replaced with the URI /customers/1/orders to find all the orders for customer 1, and then /orders/99/products to find the products in this order.

Tip: Avoid requiring resource URIs more complex than collection/item/collection/item.

Another factor is that all web requests impose a load on the web server. The more requests, the bigger the load. Therefore, **try to avoid "chatty" web APIs** that expose a large number of small resources. Such an API may require a client application to send multiple requests to find all of the data that it requires. Instead, you might want to denormalize the data and combine related information into bigger resources that can be retrieved with a single request. However, you need to balance this approach against the overhead of fetching data that the client doesn't need. Retrieving large objects can increase the latency of a request and incur additional bandwidth costs.

Avoid introducing dependencies between the web API and the underlying data sources. For example, if your data is stored in a relational database, the web API doesn't need to expose each table as a collection of resources. In fact, that's probably a poor design. Instead, think of the web API as an abstraction of the database. If necessary, introduce a mapping layer between the database and the web API. That way, client applications are isolated from changes to the underlying database scheme.

Finally, **it might not be possible to map every operation implemented by a web API to a specific resource. You can handle such non-resource scenarios through HTTP requests that invoke a function and return the results as an HTTP response message**. For example, a web API that implements simple calculator operations such as add and subtract could provide URIs that expose these operations as pseudo resources and use the query string to specify the parameters required. For example a GET request to the URI /add?operand1=99&operand2=1 would return a response message with the body containing the value 100. **However, only use these forms of URIs sparingly.**

3.3. Define operations in terms of HTTP methods

The HTTP protocol defines a number of methods that assign semantic meaning to a request. The common HTTP methods used by most RESTful web APIs are:

- 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, but typically not the URI, which is defined by the service.
- PUT either creates resources (where we assign the URI, unlike in posts) or replaces the resource at the specified URI. The body of the request message specifies the resource to be created or updated.
- PATCH (**rarely used**) 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.

The effect of a specific request should depend on whether the resource is a collection or an individual item.

The differences between POST, PUT, and PATCH can be confusing.

- A POST request creates a resource. The server assigns a URI for the new resource, and returns that URI to the client. In the REST model, you frequently apply POST requests to collections. The new resource is added to the collection. A POST request can also be used to submit data for processing to an existing resource, without any new resource being created.
- A PUT request creates a resource or updates an existing resource. **The client specifies the URI for the resource**. **The request body contains a complete representation of the resource**. If a resource with this URI already exists, it is replaced. Otherwise a new resource is created. PUT requests are most frequently applied to resources that are individual items, such as a specific customer, rather than collections. **A server might support updates but not creation via PUT**. **Whether to support creation via PUT depends on whether the client can meaningfully assign a URI to a resource before it exists**. If not, then use POST to create resources and PUT or PATCH to update.
- A PATCH request performs a partial update to an existing resource. The client specifies the URI for the resource. The request body specifies a set of changes to apply to the

resource. This can be more efficient than using PUT, because the client only sends the changes, not the entire representation of the resource. Technically PATCH can also create a new resource (by specifying a set of updates to a "null" resource), if the server supports this. **BUT don't create resources with patch**

PUT requests must be idempotent. If a client submits the same PUT request multiple times, the results should always be the same (the same resource will be modified with the same values). POST and PATCH requests are not guaranteed to be idempotent.

For an easy understanding use this structure for every resource (https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/):

Resource	**GET read**	**POST create**	**PUT update**	**DELETE**
/cars	**Returns a list of cars**	**Create a new car**	**Bulk update of cars**	**Delete all cars**
/cars/711	**Returns a specific car**	**Method not allowed (405)**	**Updates a specific car**	**Deletes a specific car**

3.4. Conform to HTTP semantics and conventions

3.4.1. GET methods

A successful GET method typically returns HTTP status code 200 (OK). If the resource cannot be found, the method should return 404 (Not Found).

3.4.2. POST methods

If a POST method creates a new resource, it returns HTTP status code 201 (Created). The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource.

If the method does some processing but does not create a new resource, the method can return HTTP status code 200 and include the result of the operation in the response body. Alternatively, if there is no result to return, the method can return HTTP status code 204 (No Content) with no response body.

If the client puts invalid data into the request, the server should return HTTP status code 400 (Bad Request). The response body can contain additional information about the error or a link to a URI that provides more details.

3.4.3. PUT methods

If a PUT method creates a new resource, it returns HTTP status code 201 (Created), as with a POST method. If the method updates an existing resource, it returns either 200 (OK) or 204 (No Content). In some cases, it might not be possible to update an existing resource. In that case, consider returning HTTP status code 409 (Conflict).

Consider implementing bulk HTTP PUT operations that can batch updates to multiple resources in a collection. The PUT request should specify the URI of the collection, and the request body should specify the details of the resources to be modified. This approach can help to reduce chattiness and improve performance.

3.4.4. DELETE methods

If the delete operation is successful, the web server should respond with HTTP status code 204, indicating that the process has been successfully handled, but that the response body contains no further information. If the resource doesn't exist, the web server can return HTTP 404 (Not Found).

3.5. Asynchronous operations

Sometimes an operation might require processing that takes awhile to complete. If you wait for completion before sending a response to the client, it may cause unacceptable latency. If so, consider making the operation asynchronous. Return HTTP status code 202 (Accepted) to indicate the request was accepted for processing but is not completed.

You should expose an endpoint that returns the status of an asynchronous request, so the client can monitor the status by polling the status endpoint. Include the URI of the status endpoint in the Location header of the 202 response. For example:

HTTP

```
HTTP/1.1 202 Accepted
Location: /api/status/12345
```

If the client sends a GET request to this endpoint, the response should contain the current status of the request. Optionally, it could also include an estimated time to completion or a link to cancel the operation.

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status":"In progress",
    "link": { "rel":"cancel", "method":"delete", "href":"/api/status/12345" }
}
```

If the asynchronous operation creates a new resource, the status endpoint should return status code 303 after the operation completes.

3.6. Filter and paginate data

Exposing a collection of resources through a single URI can lead to applications fetching large amounts of data when only a subset of the information is required. For example, suppose a client application needs to find all orders with a cost over a specific value. It might retrieve all orders from the /orders URI and then filter these orders on the client side. Clearly this process is highly inefficient. It wastes network bandwidth and processing power on the server hosting the web API.

Instead, the API can allow passing a filter in the query string of the URI, such as /orders?minCost=n. The web API is then responsible for parsing and handling the minCost parameter in the query string and returning the filtered results on the server side.

GET requests over collection resources can potentially return a large number of items. You should design a web API to limit the amount of data returned by any single request. Consider supporting query strings that specify the maximum number of items to retrieve and a starting offset into the collection. For example:

/orders?limit=25&offset=50

Also consider imposing an upper limit on the number of items returned, to help prevent Denial of Service attacks. To assist client applications, GET requests that return paginated data should also include some form of metadata that indicate the total number of resources available in the collection.

You can use a similar strategy to sort data as it is fetched, by **providing a sort parameter** that takes a field name as the value, such as **/orders?sort=ProductID**. **However, this approach can have a negative effect on caching, because query string parameters form part of the resource identifier used by many cache implementations as the key to cached data**.

You can extend this approach to limit the fields returned for each item, if each item contains a large amount of data. For example, you could use a query string parameter that accepts a comma-delimited list of fields, such as /orders?fields=ProductID,Quantity.

Give all optional parameters in query strings meaningful defaults. For example, set the limit parameter to 10 and the offset parameter to 0 if you implement pagination, set the sort parameter to the key of the resource if you implement ordering, and set the fields parameter to all fields in the resource if you support projections.

3.7. Support partial responses for large binary resources

A resource may contain large binary fields, such as files or images. To overcome problems caused by unreliable and intermittent connections and to improve response times, consider enabling such resources to be retrieved in chunks. To do this, the web API should support the Accept-Ranges header for GET requests for large resources. This header indicates that the GET operation supports partial requests. The client application can submit GET requests that return a subset of a resource, specified as a range of bytes.

Also, consider implementing HTTP HEAD requests for these resources. A HEAD request is similar to a GET request, except that it only returns the HTTP headers that describe the resource, with an empty message body. A client application can issue a HEAD request to determine whether to fetch a resource by using partial GET requests.

3.8. Use HATEOAS to enable navigation to related resources

One of the primary motivations behind REST ***WAS*** that it should be possible to navigate the entire set of resources without requiring prior knowledge of the URI scheme. Each HTTP GET request should return the information necessary to find the resources related directly to the requested object through hyperlinks included in the response, and it should also be provided with information that describes the operations available on each of these resources. This principle is known as HATEOAS, or Hypertext as the Engine of Application State. The system is effectively a finite state machine, and the response to each request contains the information necessary to move from one state to another; no other information should be necessary.

Currently there are no standards or specifications that define how to model the HATEOAS principle. The examples shown in this section illustrate one possible solution. And this tells you that this style is not used much.

For example, to handle the relationship between an order and a customer, the representation of an order could include links that identify the available operations for the customer of the order. Here is a possible representation:

JSON

```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"http://adventure-works.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
      "href":"http://adventure-works.com/customers/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"customer",
      "href":"http://adventure-works.com/customers/3",
      "action":"DELETE",
      "types":[]
    },
    {
      "rel":"self",
      "href":"http://adventure-works.com/orders/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"self",
      "href":"http://adventure-works.com/orders/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"self",
      "href":"http://adventure-works.com/orders/3",
      "action":"DELETE",
      "types":[]
    }]
}
```

In this example, the links array has a set of links. Each link represents an operation on a related entity. The data for each link includes the relationship ("customer"), the URI (http://adventure-works.com/customers/3), the HTTP method, and the supported MIME types. This is all the information that a client application needs to be able to invoke the operation.

The links array also includes self-referencing information about the resource itself that has been retrieved. These have the relationship self.

The set of links that are returned may change, depending on the state of the resource. This is what is meant by hypertext being the "engine of application state."

3.9. Use HTTP headers for serialization formats

Both, client and server, need to know which format is used for the communication. The format has to be specified in the HTTP-Header.

`Content-Type` defines the request format.

`Accept` defines a list of acceptable response formats.

3.10. Versioning a RESTful web API

It is highly unlikely that a web API will remain static. As business requirements change new collections of resources may be added, the relationships between resources might change, and the structure of the data in resources might be amended. While updating a web API to handle new or differing requirements is a relatively straightforward process, you must consider the effects that such changes will have on client applications consuming the web API. The issue is that although the developer designing and implementing a web API has full control over that API, the developer does not have the same degree of control over client applications which may be built by third party organizations operating remotely. The primary imperative is to enable existing client applications to continue functioning unchanged while allowing new client applications to take advantage of new features and resources.

Versioning enables a web API to indicate the features and resources that it exposes, and a client application can submit requests that are directed to a specific version of a feature or resource. The following sections describe several different approaches, each of which has its own benefits and trade-offs.

3.10.1. No versioning

This is the simplest approach, and may be acceptable for some internal APIs. Big changes could be represented as new resources or new links. Adding content to existing resources might not present a breaking change as client applications that are not expecting to see this content will simply ignore it.

Existing client applications might continue functioning correctly if they are capable of ignoring unrecognized fields, while new client applications can be designed to handle this new field. However, if more radical changes to the schema of resources occur (such as removing or renaming fields) or the relationships between resources change then these may constitute breaking changes that prevent existing client applications from functioning correctly. In these situations you should consider one of the following approaches.

3.10.2. URI versioning

Each time you modify the web API or change the schema of resources, you add a version number to the URI for each resource. The previously existing URIs should continue to operate as before, returning resources that conform to their original schema.

Extending the previous example, if the address field is restructured into sub-fields containing each constituent part of the address (such as streetAddress, city, state, and zipCode), this version of the resource could be exposed through a URI containing a version number, such as **http://adventure-works.com/v2/customers/3:**

This versioning mechanism is very simple but depends on the server routing the request to the appropriate endpoint. However, it can become unwieldy as the web API matures through several iterations and the server has to support a number of different versions. Also, from a purist's point of view, in all cases the client applications are fetching the same data (customer 3), so the URI should not really be different depending on the version. This scheme also complicates implementation of HATEOAS as all links will need to include the version number in their URIs.

3.10.3. Query string versioning

Rather than providing multiple URIs, you can specify the version of the resource by using a parameter within the query string appended to the HTTP request, such as http://adventure-works.com/customers/3?version=2. The version parameter should default to a meaningful value such as 1 if it is omitted by older client applications.

This approach has the semantic advantage that the same resource is always retrieved from the same URI, but it depends on the code that handles the request to parse the query string and send back the appropriate HTTP response.

3.10.4. Header versioning

Rather than appending the version number as a query string parameter, you could implement a custom header that indicates the version of the resource. This approach requires that the client application adds the appropriate header to any requests, although the code handling the client request could use a default value (version 1) if the version header is omitted. The following examples utilize a custom header named Custom-Header. The value of this header indicates the version of web API.

Version 1:

HTTP

GET http://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=1

When you select a versioning strategy, you should also consider the implications on performance, especially caching on the web server. The URI versioning and Query String versioning schemes are cache-friendly inasmuch as the same URI/query string combination refers to the same data each time.

4. API Languages

The best thing is to go by example, which is self-explanatory and becomes clear once you play with it in apiary.io (still, i'll add comments to clarify some aspects)

```
swagger: '2.0'
info:
  version: '1.0'
  title: "title of your API"
  description: Polls is a simple API allowing consumers to view polls
and vote in them.
  license:
    name: MIT
    url: https://github.com/apiaryio/polls-api/blob/master/LICENSE
host: polls.apiblueprint.org
basePath: /
schemes:
- https
consumes:
- application/json
produces:
- application/json
paths:
  /questions:
    x-summary: Questions Collection
    get:
      summary: List All Questions
      responses:
        200:
          description: Successful Response
          schema:
            type: array
            items:
              $ref: '#/definitions/Question'
          examples:
            application/json:
              - question: Favourite programming language?
                published_at: '2015-08-05T08:40:51.620Z'
                choices:
                  - choice: Swift
                    votes: 2048
                  - choice: Python
                    votes: 1024
                  - choice: Objective-C
                    votes: 512
                  - choice: Ruby
                    votes: 256
    post:
      description: >-
        You may create your own question using this action. It takes a
JSON
        object containing a question and a collection of answers in
the
        form of choices.
      summary: Create a New Question
      parameters:
        - name: body
          in: body
          required: true
          schema:
            $ref: '#/definitions/QuestionRequest'
      responses:
        201:
          description: ''
          schema:
            $ref: '#/definitions/Question'
          examples:
            application/json:
                question: Favourite programming language?
                published_at: '2015-08-05T08:40:51.620Z'
                choices:
                  - choice: Swift
                    votes: 0
                  - choice: Python
```

```
                    votes: 0
                  - choice: Objective-C
                    votes: 0
                  - choice: Ruby
                    votes: 0
definitions:
  Question:
    title: Question
    type: object
    properties:
      question:
        type: string
      published_at:
        type: string
      choices:
        type: array
        items:
          $ref: '#/definitions/Choice'
    required:
      - question
      - published_at
      - choices
  Choice:
    title: Choice
    type: object
    properties:
      votes:
        type: integer
        format: int32
      choice:
        type: string
    required:
      - votes
      - choice
  QuestionRequest:
    title: Question Request
    type: object
    properties:
      question:
        type: string
      choices:
        type: array
        items:
          type: string
    required:
      - question
      - choices
    example:
```

```
question: Favourite programming language?
choices:
  - Swift
  - Python
  - Objective-C
  - Ruby
```