ANTONIO PINTUS and FEDERICO PINNA

THE WEB API
DESIGN GUIDELINES FOR HAPPY DEVELOPERS

A short pragmatic guide to build effective and funny Web APIs

The Web API Design Guidelines for Happy Developers

A short pragmatic guide to build effective and funny Web APIs

Federico Pinna and Antonio Pintus

This book is for sale at http://leanpub.com/thewebapinntux

This version was published on 2015-12-24

Table of Contents

Introduction

When developing a modern Web Application, it’s increasingly likely that you, as a developer, will need to use an API: be it a social network plugin API, an e-commerce checkout API, a Captcha, a Web Analytics suite. In today’s Web development APIs are ubiquitous.

But as you wandered through dozens of different APIs, you’ve certainly found out already that they’re not all born equal: for each API that is well thought, well documented, easy to understand and fun to use, there are dozens that are obscure and inconsistent, error-prone and tedious.

The goal of this ebook is to provide a set of guidelines through key points/tips on how to design Web APIs that are, at the same time: nice and sound, fun to use and logically consistent. We don’t like to call them REST, as we don’t want to introduce unneeded constraints in our guidelines. We prefer to call them *Pragmatic Web APIs*.

The intended audience of this ebook are software architects and web developers, and the only prerequisite to be able to follow the exposition is to have a basic understanding of how HTTP, the main protocol of the Web, works and what a Web API is. The guide will not require nor use any particular programming language or network infrastructure: all the principles and tips described, will apply to any language suitable for Web development, as well as any modern Web server technology.

To write this book in a meaningful way we have drawn not only from our daily experience, but also from other good sources and works about the same topics. Resources and articles that we considered most useful can be found in the References section of this guide and constitute a valid and updated list to deepen the several aspects involved in API designing and/or to get a different opinion. The authors of these articles, guides, posts, blogs and websites deserve our thanks, thus go to visit and read them!

We believe that a good API is a beautiful and fun one, and in this spirit we hope you’ll enjoy this short ebook and that it will inspire you to build the next API that we’ll enjoy using.

About the Authors

Antonio Pintus

Software Architect.

15+ years of experience as Technologist in the ICT Department at CRS4, where he contributes to national and international funded research projects. Co-founder and CTO of the Internet of Things company named **Paraimpu**. His main activities and skills are in design and development of scalable Web software systems and APIs, in particular related to the Internet of Things, designing new solutions for connecting and mashuping services and smart devices through the Web. He is author and co-author of several publications and scientific papers presented in national and international conferences.

He loves APIs, JavaScript and Node.js. Amateur photographer and design lover.

Website | Blog | Twitter | Github | Instagram

Federico Pinna

Having written his first computer program at the age of nine, he’s being developing software for over thirty years. Co-founder and CTO of **Vivocha**, a startup providing a cloud based Online Engagement Platform to medium and large enterprises. Over the past twenty years he’s designed and developed countless protocol stacks and APIs, ranging from Voice Over IP and Computer Telephony Integration systems, to, more recently, secure instant messaging and WebRTC middlewares. In 2007, he’s awarded an international patent as the inventor of iSMIL, a real-time video editing system.

He loves programming languages and wandering off the beaten track to find new solutions. Music lover, lifelong gamer.

Facebook | Twitter | Github

Documentation

When developing an API, some developers start with a design document, some prefer to prepare the test tools, most go straight to the code. When using someone else’s API, though, all developers start at the same place: the documentation. And frequently it’s the quality of the documentation that sorts to good APIs from the bad ones: an API, to be complete, needs to be thoroughly documented. A Pragmatic Web API demands a pragmatic documentation, with good examples, clear use cases, with tips to avoid common pitfalls and guidelines on best practises.

In this spirit, we believe in clarity over brevity: avoid *super cryptic albeit formally unexceptionable* specifications of your APIs, and prefer instead a style more welcoming to non-experts. Every time a user doesn’t fully understand your documentation, he’s likely to either stop using your API, write buggy code or send you a support request!

There are many openly available tools that can be used to produce a good documentation web site. An absolutely not exhaustive list of valid alternatives includes:

- Github Wikis
- Slate
- Swagger (recently launched the Open API Initiative, including an API specification language and tools)

Some examples of APIs having a great documentation are:

- Stripe
- Twilio
- Twitter

Always document your API

And build a companion website with an exhaustive and navigable documentation.

Resource Names and URL Design

A resource is a domain model you need to expose with an API and it is strictly related to specific context of your application, your service, your business. A resource is any entity the users will be directly manipulating through your API. Anything can be a resource: an entity like a Book can be a resource, a User, an Order, a Post, a Ticket, a Reservation, they can all be resources. Obviously, each Resource has its own properties (fields); for example, a Book could have the isbn, author, year properties. Each Resource, through APIs has a representation, expressed in a particular format, usually: JSON or XML. For example, a Book representation in JSON could look like the following:

```
{
   "title":"Web API tips",
   "author":"Antonio Pintus",
   "isbn":"...",
   "year":"2015",
   ...
}
```

Each resource has its own unique URL.

Commonly, there are two types of resources:

- Collections, represent lists of resources. For example, Books, Users, Tickets could all represent collections. Typically all the resources in a collection have the same type and, thus, some (not necessarily all) properties in common.
- Elements are the individual members of a collection.

The canonical representation in JSON of a collection is an array of objects:

```
[
   {
     "title":"Web API tips",
     "author":"Antonio Pintus"
   },
   {
     "title":"Coffee brewing tips",
     "author":"Federico Pinna"
   }
]
```

Give each resource a unique identifier (id)

A good practise is to use a single property name to specify the identifier for all the different resource type and to always include such property when representing a resource.

Example:

```
{
  "id": "abcd",
  "title":"Web API tips",
  "author":"Antonio Pintus"
}
```

Use plural nouns for collections

Every resource is identified by a noun, use the plural form in designing URL. Don’t use verbs, verbs are bad and recall something related to old-style RPC design. (About verbs, see the **HTTP Methods** section: let HTTP methods be the verbs when modelling your APIs).

Examples:

```
/services
/connections
/orders
/books
```

Use ids to access single resources

To design an URL for a single resource, i.e. an element in a collection, use its unique *id*, in the form `/{collection}/{resource id}`.

Example:

```
/books/abcd
/services/123456789
/users/john_doe
```

API Base URL and Versioning

API base URLs must contain versioning and should be simple, readable and easy to understand. Basically it should reflect your core business concept and platform. The base URL can be as simple as a dedicated domain name plus a version info, or a more structured path, allowing you to serve totally different APIs from the same domain name, each with its own version information.

Examples:

```
https://api.paraimpu.com/v1
https://api.vivocha.com/v5
https://api.paraimpu.com/v1/things
https://api.vivocha.com/v5/payments
https://www.example.org/api/some-api/v1
https://www.example.org/api/some-api/v2
https://www.example.org/api/other-api/v12
```

Sometimes, providing aliases for some special versions can be handy:

```
https://api.example.org/latest
https://api.example.org/beta
```

Always use API versioning in base URLs

Versioning becomes indispensable to not breaking your users when you will release a new version and gives you the chance to deprecate an older API version, to document it, alert developers without immediately breaking things. Don’t accept any requests that do not specify a version number.

Use a consistent versioning scheme

Version information can be expressed in many ways: it’s common to see versioning specified as a date (`api.example.org/2015-06-15`) or as a number (`api.example.org/v3`). It’s important to use a consistent way and to give means to the users to determine which version is newer: to that extent, using version names is not very advisable (is `api.example.org/fuzzy-bear` newer than `api.example.org/dizzy-pigeon`?).

When the underlying application is using SemVer for its versioning, using the major version in the API base URL is a very good idea.

HTTP Methods

The HTTP protocol provides all the needed methods to suitably implement the CRUD (Create Read Update Delete) principle just out-of-the-box: each of the basic functions can and should be mapped to the correct HTTP method. Use `POST` to create, `GET` to read, `PUT` to update and `DELETE` to (guess what!) delete a resource.

Create with POST

To create a new resource on the server, your API should require a `POST` on the collection the new resource will belong to. The body of the request should contain the complete object you want to create, with the possible exception of the id field, which could be allocated by the server.

For example, to a create a new Book you would expect a request like this:

```
POST /v1/books HTTP/1.1
Host: https://api.example.org
Content-Type: application/json

{
  "title": "Cryptonomicon",
  "author": "Neal Stephenson"
}
```

Other examples:

- `POST /2015-06-15/things`, create a new Thing from body data
- `POST /v3/services`, create a new Service from body data
- `POST /v1/things/123`, **WRONG!** Use POST only on collections!

A `POST` request should return:

1. a status code of `201` to confirm the creation
2. the `Location` HTTP header with the URL of the newly created resource
3. the new resource representation in the body of the response

For example, the response to our above request to create a Book, should look like:

```
HTTP/1.1 201 Created
Location: https://api.example.org/v1/books/abcde
Content-Type: application/json
```

```
{
  "id": "abcde",
  "title": "Cryptonomicon",
  "author": "Neal Stephenson"
}
```

Read with GET

To retrieve a resource representation use the `GET` method. The `GET` method must not modify the resource on the server and it can be invoked on both collection and element URLs: in the former case all the accessible elements are returned, in the latter only the requested resource is returned

Example of a request to a collection URL:

```
GET /v1/books HTTP/1.1
Host: https://api.example.org


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

[
  {
    "id": "abcde",
    "title": "Cryptonomicon",
    "author": "Neal Stephenson"
  },
  {
    "id": "fghij",
    "title": "Cloud Atlas",
    "author": "David Mitchell"
  }
]
```

Example of a request to an element URL:

```
GET /v1/books/abcde HTTP/1.1
Host: https://api.example.org


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

{
  "id": "abcde",
  "title": "Cryptonomicon",
  "author": "Neal Stephenson"
}
```

Update with PUT

To update an existing element in a collection, or even an entire collection at one, on the server, provide the `PUT` method. In order to avoid ambiguities **DON'T** allow API users to create a new resource using `PUT`, use `POST` instead.

In case of success, return a representation of the updated resource in the response body.

Some examples:

- `PUT /v1/things/123`, update the Thing with id 123 with the data in the body
- `PUT /v1/things`, update several Things at once! The body must contains an array with all the elements to update.

Here's a complete example:

```
PUT /v1/books/abcde HTTP/1.1
Host: https://api.example.org
Content-Type: application/json

{
  "title": "Snow Crash",
  "author": "Neal Stephenson"
}


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

{
  "id": "abcde",
  "title": "Snow Crash",
  "author": "Neal Stephenson"
}
```

Some specifications, like JSON:API, suggest the use of the `PATCH` method to update a resource. We're cool with it.

Other APIs use `PATCH` to request a partial update of a resource.

Delete with… DELETE

To delete an existing resource on the server provide the `DELETE` method.

Examples:

- `DELETE /v1/things/123`, delete the Thing with id 123

- `DELETE /v1/things`, delete ALL the accessible Things: use at your own risk!

Limitations of HTTP PUT e DELETE

When HTTP `PUT` and `DELETE` methods are not supported by the API consumers you need to provide a way to use them. This kind of limitations, while increasingly rare, can still be found on very lightweight or very old browsers, and in particular in libraries targeted to small devices.

A possible solution to this problem is to provide an alternative way of calling the `PUT` and `DELETE` methods using the `X-HTTP-Method-Override` header in a `POST` request. Avoid using `GET` in this scenario, as `GET` mandates idempotency and `PUT` and `DELETE` aren’t.

Examples:

```
POST /v1/things/123 HTTP/1.1
X-HTTP-Method-Override: PUT
Host: https://api.example.org
{...}


POST /v1/things/123 HTTP/1.1
X-HTTP-Method-Override: DELETE
Host: https://api.example.org
{...}
```

But, in order to be even more pragmatic, just like we like it over here… here’s another tip ;)

Provide PUT and DELETE methods call alternative using a POST and specifying a `method` URL parameter

Examples:

- `POST /v1/things/123?method=put`
- `POST /v1/things/123?method=delete`

Caching

To improve the performance of you API and to reduce the bandwidth required to respond to your users, you should consider using `Etag` or `Last-Modified` headers to support caching.

Include an `ETag` or a `Last-Modified` HTTP header in all responses

They involve a little different mechanism (returning hashes or checksums of the resources or timestamps, respectively). API consumers should be able to check for resource staleness in subsequent calls by supplying values through the `If-None-Match` or `If-Not-Modified-Since` request headers.

Rate Limiting

To avoid API call abuse and server overload, you should limit the number of requests to a particular endpoint a client can do in a given time interval. This technique, usually called *Rate Limiting*, consists in limiting requests from clients to protect the health of the service and maintain high service quality for other clients. Use the `429 Too Many Requests` HTTP Status code when rate-limiting is triggered and, as a best practise, consider adding the following header to all your responses to tell the consumer more information about API rate limits (like Twitter API does):

- `X-Rate-Limit-Limit`, the maximum rate limit for that given request (i.e. 30 requests per hour)
- `X-Rate-Limit-Remaining`, the number of requests left for established time interval (i.e. 10 requests left in the current hour)
- `X-Rate-Limit-Reset`, the remaining time before the rate limit resets, expressed in UTC epoch seconds

Repeatedly exceeding the allocated limits, can cause the offending user account to be black-listed and/or notified of the problem. Depending on the use case, you might even want to charge the user for the extra costs incurred.

Actions

The good, old HTTP methods are enough to model our web APIs following a CRUD paradigm, but… what about expressing other actions on resources? For example, think for a moment about a resource that can be activated or deactivated (start, stop or on, off). Yes, sure, you may argue that a resource can have a status, which can be updated with a `PUT`, like calling:

```
PUT /v1/things/2 HTTP/1.1
Host: https://api.example.org
Content-Type: application/json

{
  "status": "ON",
  {...}
}
```

While, generally speaking, you should avoid designing you API endpoint with support for unnecessary non-standard actions, there are scenarios that can’t be properly handled updating an existing resource, in particular when the action itself might result in the creation of a new, possibly unrelated, resource. For example, a Flights collection, representing details for an airlines flight schedule, could support a *book action*, which would end up creating a Booking object.

When special actions are indeed needed, you can model them introducing a “standard” prefix, to clearly highlight their role and peculiarity.

For example, the booking example described above, could be supported using a special `actions` namespace in the URL:

```
POST /v1/flights/12345/actions/book HTTP/1.1
Host: https://api.example.org
Content-Type: application/json

{
  "passenger": "Thaddeus Jones"
}


HTTP/1.1 201 Created
Location: https://api.example.org/v1/bookings/abcde
Content-Type: application/json

{
  "id": "abcde",
    "flightId": "12345",
    "passenger": "Thaddeus Jones"
  }
```

Use a consistent way to represent the actions

A suitable template could be:

`POST /{collection}/{resource id}/actions/{action name}`

For example:

- `POST /v1/things/4/actions/activate`
- `POST /v1/motors/1/actions/stop`

Errors and Statuses in Responses

Errors happen all the time: incomplete requests, wrong parameters, unauthorised accesses and, of course, those server bugs. Any good API must have a clear and consistent way to report errors back to its users. Web APIs leverage the HTTP protocol, that already defines all the necessary tools to effectively represent response statuses and error codes. It is strongly recommended to not introduce new custom error codes, unless absolutely necessary and to avoid using the existing one in a wrong way. For example, status `200` means “OK” and it must always be used to indicate, in a response to the API consumer, that its request was successful. Using `200 OK` to transmit error messages is as wrong as it is, unfortunately common. It’s a very bad practise that breaks all sorts of things: for instance, successful responses are often cached by network elements, like cache and proxy server, thus causing temporary error conditions to linger on the network more than needed.

The use of `200 OK` response to convey error responses was necessary back in the day when JSONP was the only way to support Cross-Origin calls to web APIs: today JSONP is mostly a thing of the past and so must be `200 OK` error messages.

Let us repeat ourselves, **don’t do this**, it’s BAD, BAD, BAD:

```
GET /v1/things/bad_id HTTP/1.1
Host: https://api.example.org


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

{
  "error": "404",
  "message": "Not Found"
}
```

Use conventional standard HTTP Response Codes

In general:

- `1xx` codes mean nothing (not true, but yeah!)
- `2xx` codes mean success
- `3xx` codes mean user want the user to repeat the request somewhere else or to used a cached response
- `4xx` codes mean it’s the users fault, a different request might succeed
- `5xx` codes mean it’s your fault, there’s nothing the user can do to get a better answer

In case of error, return also a JSON data body in responses containing a more detailed error description and info about the error. This helps developers and API consumers to better understand what happened. A more detailed error description and info about the error can be a link to a specific documentation page.

For example, a JSON format of body for error responses could be:

```
{
  "error": 404,
  "errorMessage": "Resource not found, maybe you specified a wrong id for the
source",
  "moreInfo":"https://docs.paraimpu.com/v1/support/404"
}
```

Data Formats

According to the HTTP specification, the format of the resources representation should be negotiated using the `Accept` header in client requests, specifying a suitable Media Type. When the header is omitted, the server is free to select whichever format it prefers: we like APIs that prefer the JSON format!

Examples:

```
GET /things/123 HTTP/1.1
Host: https://api.example.org
Accept: application/json


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

{
  "id": "123",
  "someProperty": "someValue"
}
```

```
GET /things/123 HTTP/1.1
Host: https://api.example.org
Accept: text/xml


HTTP/1.1 200 OK
Content-Type: text/xml

<Thing>
  <id>123</id>
  <someProperty>someValue</someProperty>
</Thing>
```

Allow selecting the response format in the URL

To facilitate developers, use a “dot-based notation” to specify the resource representation format type, appending it at the end of the endpoint URL

Thus, the previous examples become equivalent to the following requests:

- `GET /v1/things/123.json`
- `GET /v1/things/123.xml`

Always use JSON as the default format

JSON is easy to parse, Javascript native, easy to read. Some APIs still support XML, which is verbose, hard to parse and to read. Modern, new web APIs can safely avoid it at all.

Use CamelCase convention for names

Use CamelCase as the convention to express composite names in resource fields representations. Adopting CamelCase makes it easy to conform to Javascript code conventions, and a `JSON.parse()` on a resource representation doesn’t alter that.

Examples: `dateModified`, `createdAt`, `camelCaseIsGoodByDoNotOverUserIt` (bonus stealth tip!)

Dates and Time

Whenever your API needs to represent dates and time, it should do so using the ISO 8601 format. This is not only a standard, but a good one: it’s human readable and it safely bears all the information needed to accurately represent any date in any timezone.

Represent date/time in ISO 8601 format

The format is: `yyyy-MM-ddTHH:mm:ss.SSSZ`

Example:

```
{
  "createdAt": "1974-06-15T21:40:01.000Z"
}
```

Querying Resources

If you want your APIs to allow queries on your collections to search for a subset of its contents, you should to do through URL parameters: allow API users to search on a particular resource using a `GET` and specifying query terms as URL parameters

Some examples:

```
    - GET /lamps?status=ON
    - GET /documents?policy=public&owner=pintus
    - GET /posts?tags=js,programming,tips
```

Pagination

It’s a good practice to always paginate the results for resource collections. It allows a better API consumption avoiding to overload servers, database engines and network and to obtain further resources on-demand.

To provide a good pagination technique, your API should:

- Set a default and a maximum length for the number of returned results
- Support a query parameter to specify the number of returned results (e.g. `limit`)
- Support a query parameter to specify from which element the data should be returned (e.g. `skip`)
- Provide metadata about the total number of matching results and the index of the first returned element. The number of returned elements can usually be safely omitted, as it can be inferred from the body.

The metadata belongs in the HTTP header

Whenever possible, avoid using the body of the response to transmit metadata: the header is where the metadata is supposed to be.

Example:

```
GET /v1/books?limit=2&skip=5 HTTP/1.1
Host: https://api.example.org


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

```
Results-Matching: 120
Results-Skipped: 5

[
  {
    "id": "abcde",
    "title": "Cryptonomicon",
    "author": "Neal Stephenson"
  },
  {
    "id": "fghij",
    "title": "Cloud Atlas",
    "author": "David Mitchell"
  }
]
```

RFC5988 introduces a new HTTP header, `Link`, that can be used to provide a convenient way to specify direct links to more results:

```
HTTP/1.1 200 OK
Content-Type: application/json
Results-Matching: 120
Results-Skipped: 5
Link: <https://api.example.org/v1/books?limit=2&skip=7>; rel="next";

{...}
```

If you can’t put metadata in the header…

… be pragmatic and put in the body :)

In any case keep data and metadata separated!

Examples:

```
{
  "results": [
    {
      "id": "abcde",
      {...}
    },
    {
      "id": "fghij",
      {...}
    }
  ],
  "_metadata": {
    "matching": 120,
    "skipped": 5
  }
}

{
  "results": [
    {
      "id": "abcde",
      {...}
    },
    {
      "id": "fghij",
      {...}
    }
  ],
  "_metadata": {
    "totalCount": 120,
    "limit": 25,
    "skip":0
  }
}
```

Partial Responses

Sometimes, GETting a full representation of a resource can be excessive or redundant. For example, you might need only a particular field, like a profile image URL for a User or the status of a Lamp or the title (only that) of a Book.

Supporting a `fields` URL parameter is a good pragmatic solution to this problem: it allows API users to request only the desired properties of a resource representation, being more concise and saving bandwidth.

Examples:

```
GET /poi/5?fields=name HTTP/1.1
Host: https://api.example.org


HTTP/1.1 200 OK
Content-Type: application/jsonA>
{
     "name": "Elephant Tower"
}
```

```
GET /services/1234-abc?fields=id,name,status HTTP/1.1
Host: https://api.example.org


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

{
   "id":"1234-abc,
   "name":"Thermostat",
   "status":"ON"
}
```

Security

All API endpoints should be available only via the HTTPS protocol, thus all API calls should use HTTPS. For particular low-profile devices, and where HTTPS is not available, some API endpoints can also support HTTP, but its use is highly discouraged.

All API request must be made over SSL. Any non-secure requests must be rejected with a `426 Upgrade Required` error:

```
HTTP/1.1 426 Upgrade Required
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
```

The server should support **HSTS (HTTP Strict Transport Security)** to instruct users to never try insecure connections to it. Server to server requests (e.g. outgoing web hooks) must also use SSL.

Authentication and Authorization

While it is ok for some APIs to be accessed by anonymous users, most of the times some form of authentication is required. There are many valid and secure authentication and authorization frameworks, greatly varying in features and complexity, ranging from basic HTTP authentication to complete OAUTH 2.0 infrastructures.

For many use-cases, adopting *Bearer Authorization* (*Bearer Token* usage) over HTTPS is a valid and pragmatic solution. The basic pre-requisite to able to use the mechanism known as Bearer Authorization is that your API users must have access to a secured control panel and must be able retrieve a unique API token/key. Other than than, all that is required is that each API request must contain that API authorization token (let's call it *API_KEY*).

According to the features of your device or your programming language, you have two choices to send the user token in an API call:

1. Add to the call request the HTTP `Authorization: Bearer API_KEY` header:

   ```
   GET /v1/things HTTP/1.1
   Host: https://api.example.org
   Authorization: Bearer abcd-123-efg-456
   ```

2. alternatively, use a URL query parameter, naming it `access\_token` with *API_KEY* as value. In this case, the Authorization HTTP Header isn't used.

   ```
   GET /v1/things?access_token=abcd-123-efg-456 HTTP/1.1
   Host: https://api.example.org
   ```

⚠ An *API_KEY* is intended to be kept private and should not be shared in any way.

Server to server requests should use also use authentication and authorization. In addition to what we've just discussed, you should require valid client certificates and a HMAC signature on each request.

Browser Same Origin Policy

In order to allow to client-side JavaScript scripts to directly call our APIs (i.e., from browsers) we need to adopt a mechanism to allow calls from scripts that are “cross-origin”, in other words: which are not downloaded from the same origin/website of the APIs.

A recommended mechanism is **CORS (Cross-Origin Resource Sharing)** and generally it replaces the old JSONP mechanism, which we can finally consider obsolete.

CORS is a mechanism that allows many resources on a Web page to be requested from another domain outside the domain from which the resource originated and it is the preferred technique when dealing with the same-origin policy “issue”. CORS is a recommended standard by W3C.

A jQuery-esque use of an CORS-compatible Web API directly from web page looks something like this:

```
$.ajax({
   type: "PUT",
   url: "https://api.example.org/v1/lamps/123?access_token=1234-abcd-gg-3456",
   contentType: "application/json",
   data: JSON.stringify({ "status": "ON" }),
   success: function(response) {
      console.log(JSON.stringify(response));
   },
   error: function() {
      console.log("An error occurred");
   }
});
```

Implement support for CORS *pre-flight* requests

If possible, implement the full CORS specification to allow Cross-Origin Resource calls to your API, including *pre-flight*. Pre-flight allows client to “probe” your server, to check whether a request would be accepted, without actually making the request (which might imply posting significative amounts of data). This is accomplished by sending an `OPTIONS` request before the actual method call.

For example, to check is the server would allow a `DELETE` request from a particular Cross-Domain origin:

```
OPTIONS /v1/things/abc
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: origin, x-requested-with, accept
Origin: http://cool.api.user.com
```

A successful response would look like this:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://cool.api.user.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
```

Software Development Kits

You know, we (developers) are lazy.

Lazy, yes, enthusiasts and lazy. We would want to discover new APIs and to immediately start to use them to build our new applications. Thus… make a developer (really) happy:

Provide an SDK with libraries and modules to talk with your APIs.

Support the most used programming languages and (why not?) release them as open-source.

It will accelerate the adoption of your API and engage communities of developers and… it gives you an idea of how funny and sweet your APIs are.

And don’t forget to document your SDKs!

Conclusions

Today, web-based APIs are the bricks of modern, distributed, scalable, amazing new applications. APIs open to developers an infinite range of opportunities: from rapid prototyping to new useful production-ready products. In the last years, REST like approaches have been the most used in shaping and designing new APIs, after the “Rise and the Fall of Ziggy SOA and the Spiders from XML” ;)

REST principles are at the same time easy and somewhat rigid. We believe in a less orthodox approach in designing and implementing web APIs. They should be easy, reasonable, predictable and fun to use, well documented and respectful about web and HTTP standards. This short guide listed the key points about our view toward pragmatic web APIs.

Thank you,

Antonio

Federico

And… one more tip

Don’t forget to have fun! ;)

References

Apigee, Web API Design, https://pages.apigee.com/web-api-design-website-h-ebook-registration.html

GoCardless, Building APIs: lessons learned the hard way, https://gocardless.com/blog/building-apis

JSON:API, http://jsonapi.org

White House Web API Standards, https://github.com/WhiteHouse/api-standards

HTTP API Design Guide, https://github.com/interagent/http-api-design

HTTP API Design Standards, https://github.com/gocardless/http-api-design/blob/master/README.md

Vinay Sahni, Best Practices for Designing a Pragmatic RESTful API, http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

Josh Wyse, 418: I’m a Teapot, and other bad API responses, http://blog.cloud-elements.com/418-im-a-teapot-and-other-bad-api-responses-restful-api-design

HTTP 1.1 Protocol, http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Web Linking, https://tools.ietf.org/html/rfc5988

Cross-Origin Resource Sharing, http://www.w3.org/TR/cors/