The Design of Web APIs

Arnaud Lauret

Foreword by Kin Lane

MANNING

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Jenny Stout
Technical development editor:	Michael Lund
Review editor:	Ivan Martinović
Production editor:	Deirdre Hiam
Copy editor:	Frances Buran
Proofreader:	Melody Dolab
Technical proofreader:	Paul Grebenc
Typesetter:	Happenstance Type-O-Rama
Cover designer:	Marija Tudor

ISBN 9781617295102
Printed in the United States of America

11.3 Choosing an API style according to the context

When you've mastered or are used to using a tool like a hammer, it's very tempting to treat all problems like nails. This is a cognitive bias called the law of the instrument, the law of the hammer, or Maslow's law (https://en.wikipedia.org/wiki/Law_of_the_instrument). Such a bias can also have another effect: screwdriver users might think that a screwdriver is a better tool than a hammer, while hammer users might think the opposite. This could be called the *fannish folk law*.

But a hammer will not solve all problems, and a screwdriver is not better than a hammer; each tool is as useful as the other, but in different contexts. This book is about web API design, not carpentry or woodworking, but the same concerns apply in the tech industry too. Choosing which tool(s) you will use to design a remote API must not be done based on what you are used to, what is fashionable, or your personal preference; it must be done according to the context. And being able to choose the right tool requires you to know more than one.

Web APIs can easily be reduced to unitary and synchronous request/response + REST + HTTP 1.1 + JSON web APIs, which is nowadays one of the most commonly used ways to enable software-to-software communication in order to expose goals fulfilling targeted users' needs. Therefore, API designers could be tempted to use this set of tools in all situations, in all contexts. In this book, this toolset is only used to expose fundamental API design principles that you can use when designing other types of remote APIs.

We've already discovered some other tools that can be added to our toolboxes to be used in the appropriate contexts. In section 6.2.1, for example, you saw that JSON was not the only possible data format for APIs; you can use XML, CSV, PDF, or many other formats. You also saw in section 11.2.1 that sometimes it might even be counterproductive to use JSON in a context where consumers are used to an existing standardized XML format. In section 10.3.6, you learned that REST APIs are not the only option when creating web APIs. Using a query language might bring more flexibility when requesting data (but less caching possibilities). In section 11.1, you discovered that a synchronous request/response consumer-to-provider mechanism is not the only way of enabling communication between two systems. We can create asynchronous goals, notify consumers of events, stream data, and even process multiple elements in one call. And in section 10.2.1, you learned that HTTP 2 can be used instead of the good old HTTP 1.1 protocol.

We already know that context plays an important role in the choice of tools, and we already know about several different tools. But as API designers and software and systems designers, in general, we need to broaden our perspective in order to be sure to avoid the law of the instrument. In order to do so, we will explore some alternatives to REST APIs and web APIs in this section.

11.3.1 Contrasting resource-, data-, and function-based APIs

At the time of this book's writing, there are three main ways of creating web APIs: REST, gRPC, and GraphQL. Will they still be there in five or 10 years? Will they still be the same? Only time will tell.

Is one of them better than the others? No! It depends on needs and context. The approaches shown in figure 11.10 represent three different visions of APIs: REST is resource-oriented, gRPC is function-oriented, and GraphQL is data-oriented, and each of these has its pros and cons.

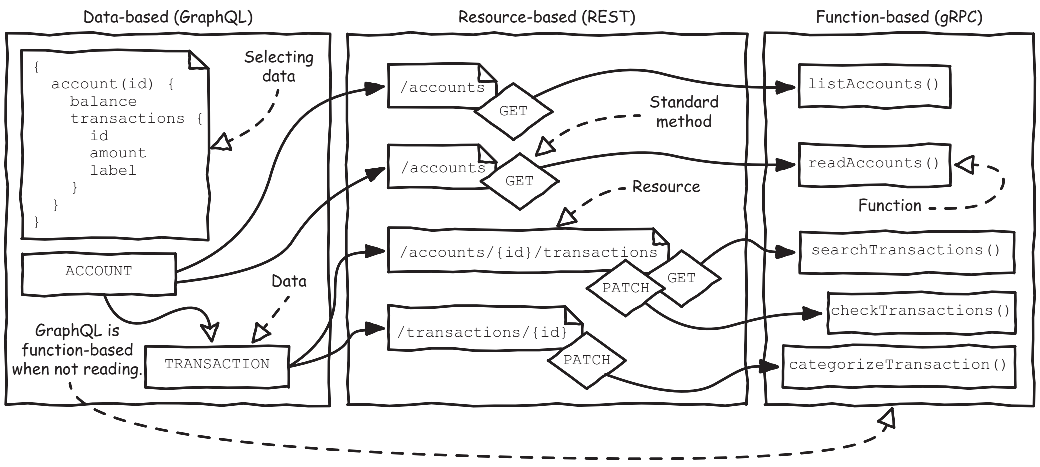

Figure 11.10 Contrasting resource-, data- and function-based APIs

You should know by now what a REST API is. As you have seen throughout this book, and especially in section 3.5.1, a REST API—or RESTful API—is an API that conforms (or at least tries to conform) to the REST architectural style introduced by Roy Fielding.[4] Such an API is resource-based and takes advantage of the underlying protocol (the HTTP protocol, in this case). Its goals are represented by the use of standard HTTP methods on resources with the results being represented by standard HTTP status codes.

In the Banking API, reading an account's details could be represented by a `GET /owners/123` request, returning a `200 OK` HTTP status along with all the customer's data if this 123 owner exists or a `404 Not Found` HTTP status if not. Updating the same owner's VIP status could be done with a `PATCH /owners/123` request, whose body would contain the new value.

Relying on an existing protocol favors consistency and makes APIs predictable, as you saw in section 6.1. Indeed, upon seeing any resource, a consumer might try to use the `OPTIONS` HTTP method to determine what can be done with it, or even try the `GET`

[4] See his PhD dissertation "Architectural Styles and the Design of Network-Based Software Architectures" at https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

method to read it or `PUT` or `PATCH` to update it. Even the most obscure `4XX` HTTP status code will be understood as an error on the consumer's end by any consumer. Such an API can also take advantage of all the existing features of HTTP, such as caching and conditional requests; designers do not have to reinvent the wheel. Server-to-consumer streaming capabilities can be added too, using SSE (see section 11.1.3). But this does not make the design of the API simple.

You have seen throughout this book that even if the HTTP protocol provides some kind of framework, it does not magically prevent us from creating terrible REST APIs. It is still up to designers to choose resource paths (`/owner` or `/owners`?) and to decide how to represent data, provide informative feedback on errors or successes beyond HTTP statuses, and more.

The gRPC framework was created by Google. The *g* stands for Google and *RPC* stands for Remote Procedure Call. An RPC API simply exposes functions.

In a function-based API, reading the 123 owner could be done by calling the `readOwner(123)` function, and updating that owner's VIP status could be done by calling `updateOwner(123, { "vip": true })`. The gRPC framework uses the HTTP 1.1 or 2 protocol as a transport layer, without using its semantics. It does not provide any standard caching mechanism. Note that it can take advantage of the HTTP 2 protocol to propose bidirectional and streaming communication. It can also use the Protocol Buffer data format, which is less verbose than XML or JSON (you can also use this format in a REST API).

Whereas in a resource-based API case, the underlying protocol provides some kind of framework, especially to describe what kind of action is being taken and what the result is, in a function-based API, it is usually up to the designers to choose their own semantics for almost everything. So, how would you represent a goal such as list owners? Should it be a `listOwners()`, `readOwners()`, or `retrieveOwners()` function? The same goes when it comes to modifying data. Should the API provide a `saveOwner()` or `updateOwner()` function?

For errors, the gRPC framework provides a standard error model including a few standard codes that map to HTTP status codes (https://cloud.google.com/apis/design/errors). For example, when calling `readOwner(123)`, a `NOT_FOUND` code (mapping to a `404 Not Found` HTTP status) can be returned along with an `Owner 123 does not exist` message. The error model can be completed with additional data in order to provide more informative feedback. As with a REST API, it is up to the designers to choose how to do that (see section 5.2.3) and also how to represent data.

We covered GraphQL briefly in section 10.3.6; it's a query language for APIs created by Facebook. A GraphQL API basically provides access to a data schema allowing consumers to retrieve exactly the data they want. It is protocol-agnostic, meaning that any protocol that lets us send requests and get responses could be used; but because the HTTP protocol is the most widely adopted, it usually is the chosen one.

Like gRPC, GraphQL does not provide any standard caching mechanism. A `POST /graphql` request with the `{ "query": "{ owner(id:123) { vip } }" }` query in its body would only return owner 123's VIP status. And when it comes to creating or updating data, GraphQL behaves like any RPC API. It uses functions that are called *mutations*. Updating owner 123's VIP status would require us to call the `updateOwner` mutation, which takes the owner's ID and an `owner` object containing the new VIP status.

GraphQL also comes with a standard error model that can be extended. Listings 11.9 and 11.10 show a query and a response with a standard error, respectively.

Listing 11.9 A GraphQL query

```
{
  owner(id: 123) {
    vip
    accounts {
      id
      balance
      name
    }
  }
}
```

Listing 11.10 A GraphQL response with an error

```
{
  "errors": [
    {
      "message": "No balance available for account with ID 1002.",
      "locations": [ { "line": 6, "column": 7 } ],
      "path": [ "owner", "accounts", 1, "balance" ]
    }
  ],
  "data": {
    "owner": {
      "vip": true,
      "accounts": [
        {
          "id": "1000",
          "balance": 123.4
          "name": "James account"
        },
        {
          "id": "1002",
          "balance": null,
          "name": "Enterprise account"
        }
      ]
    }
  }
}
```

Points to error in query

Indicates the result property affected by the error

The actual property affected by the error

The query shown in listing 11.9 requests owner 123's VIP status and account IDs, balances, and names. Unfortunately, as shown in listing 11.10, the balance could not be retrieved for the second account. The standard error model contains, for each error, a human-readable `message`, the possible sources of the error in the GraphQL query in `locations` (`balance` is on the sixth line and starts at the seventh character of the query), and the optional `path` of the affected property in the returned `data` (the `null` balance is in `data.owner.accounts[1].balance`).

Such an error seems to be the provider's fault and not the consumer's, but this is not indicated. It's up to the designers to choose how to add information to this standard error model in order to provide fully informative feedback. And obviously, like in REST and gRPC APIs, it's up to the designers to choose how to design the data model.

From a design perspective, we can see that these three different ways of creating APIs have three different ways of envisioning representations of an API's goals: resources (REST), functions (gRPC and also creations and modifications in GraphQL), and data (reads in GraphQL). Fundamentally, representing any read goal is possible in any of these API styles. When it comes to create, modify, delete, or do goals, they can be represented by a resource/method couple or a function. Each approach comes with more or less standardized elements favoring consistency and, hence, facilitating usability and design.

NOTE An API strictly following the underlying protocol's rules is the most consistent one out of the box.

But whatever the provided framework, designers still have a lot of work to do in order to design decent APIs. Regardless of the API style they choose, designers still have to identify users, goals, inputs, outputs, and errors, and choose the best possible consumer-oriented representations while avoiding the provider's perspective.

From a technical perspective, we have three different API tools or technologies that can be used over the HTTP protocol. The use of the HTTP protocol is important because it is widely accepted, and you usually do not need many, if any, modifications to your infrastructure to host or use an HTTP-based API. There are some differences between the three tools, however.

REST APIs rely on the HTTP protocol and can benefit from features such as content negotiation, caching, and conditional requests. GraphQL and gRPC do not provide such mechanisms but have some other interesting features. Thanks to the use of HTTP 2 and the ProtoBuf data format, gRPC-based APIs can provide high performance. They also provide streaming and bidirectional communication between consumer and provider. (Note that REST APIs can provide one-way streaming from provider to consumer with SSE.) And as seen in section 10.3.6, GraphQL's querying capabilities let consumers get all the data they want, and only the data they want, in a single request, but at the expense of caching capabilities.

Concerning the provider's context and especially the implementation, you obviously don't have much control over the queries that could be made by consumers in a data-based API. In non-infinitely-scalable systems, too many complex requests could result in a load higher than the underlying systems can support and terribly long response times if the implementation is not ready to prevent that. With a resource- or function-based

API, it is quite easy to avoid such problems. Because each goal's behavior is usually predictable, the solicited systems are known, and rate limiting can be used to protect the underlying systems. You can specify that each consumer cannot make more than *x* requests per second on the API, and even specialize this rate limiting by consumer and/or goal.

For data-based APIs, you could limit the number of queries or their size, but that would be pointless because it would not prevent unexpectedly complex queries from being made. You could limit the number of nodes in a request (containing one or more queries) or accept only preregistered requests, but that would be done at the expense of flexibility, making the data-based API choice almost useless. In all cases (REST, gRPC, GraphQL), a good practice would be to limit the number of items returned by default in lists.

So, which approach should you use? Such a choice cannot be made prior to analyzing your context and needs. Once you know who your consumers are and understand their contexts, the goals they need, and how they will be used, and you understand the provider's context, you can choose what kind of API will be the most appropriate. Although each context will be different, nowadays the rule of thumb is to choose REST by default. If there are very specific needs that cannot be fulfilled by a well-designed REST API, you might want to try GraphQL or gRPC.

Choosing REST by default could be seen as an example of the law of the instrument or fannish folk law, but the REST approach is capable of fulfilling most needs. It is the most widely adopted way of creating APIs, and most developers are used to it (remember section 11.2.1). Choose GraphQL for private APIs in mobile environments only if a well-designed REST API hosted in a well-configured environment is not possible (see section 10.2), and if

- You actually need advanced querying capabilities.
- You do not plan to make your API public or share it with partners.
- You do not care about caching.
- You are sure to be able to protect the underlying systems through the implementation or through infinite scalability.

Finally, choose gRPC APIs for internal-application-to-internal-application communication only if milliseconds really matter, if you do not care about caching or you are willing to handle it without relying on HTTP, and if you do not plan to make the API public or share it with partners. Also bear in mind that this choice might not be exclusive. You have already seen in section 10.3.8 that different layers of an API can fulfill different needs. Building a mobile BFF exposing a GraphQL API or a more specialized REST API is totally legit. An application can also expose a gRPC interface for internal consumers and a REST interface for external ones.