PROWESS SOFT

PROWESS SOFTWARE SERVICES

WHITEPAPER

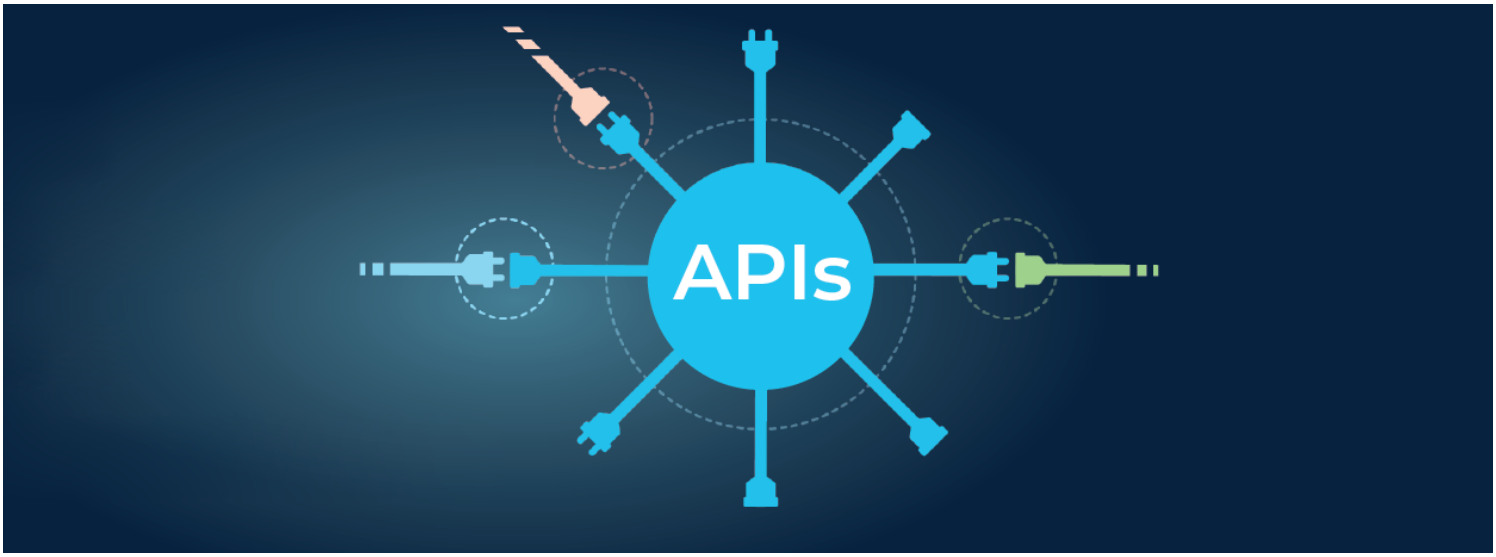
API DESIGN - Best Practices

A Guide to API Design Best Practices

Abstract

The document focuses on the best practices to consider for API design by breaking it into its basic building blocks. Along with each recommendation the document also describes the reason and impact of a certain design decision through scenarios. The document provides recommendations on object representations, usage of headers and URL/URI templates to consider.

CONTENTS



API Management- Introduction & Scope

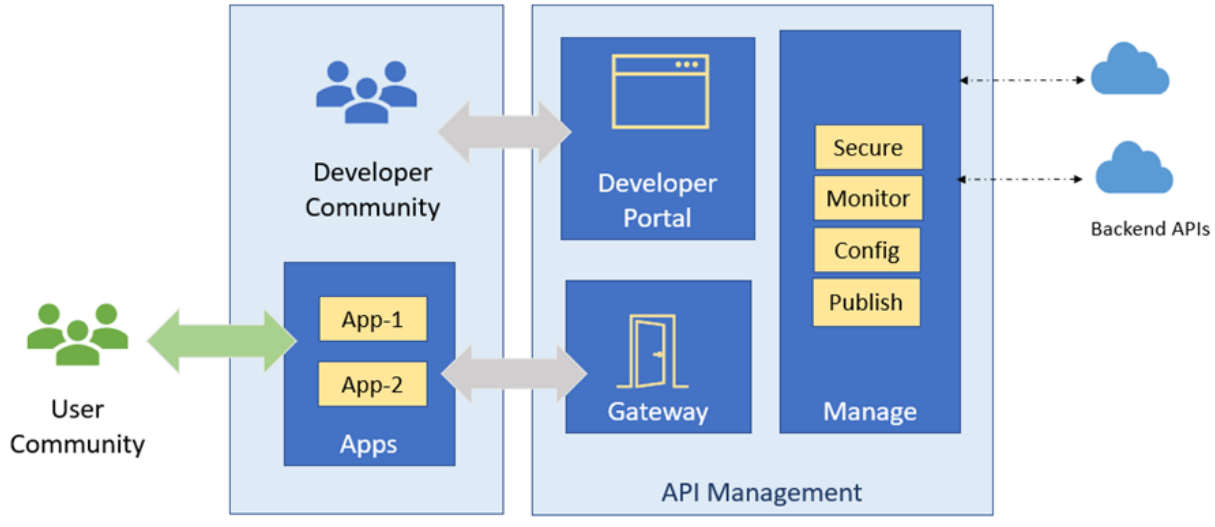
With the onset of digital transformation journeys for enterprises across the world, APIs have become a natural strategy for enterprises to provide access to their products and services. APIs enable easy communication and integration of various systems that help accomplish enterprise business goals. What was once a sizeable undertaking of building custom code that involved understanding the internals of other systems, building, and configuring the adapters, has now become an API-based integration that is both easy and familiar for the developer community to integrate with. It is safe to say that if API is not part of the strategy for the product or a service, the real business value of the product may never be realized.

The wide adoption of APIs may be hugely attributed to the ease and familiarity of APIs within the developer community. Most of the REST-based features on which APIs are designed, come with the HTTP protocol, which is widely used as the transport protocol with APIs.

Representational State Transfer (REST) is an architectural style for developing web APIs. It defines a set of constraints on how distributed hypermedia systems should be managed for data exchange. HTTP is an application layer protocol that is built upon the principles of REST for distributed hypermedia systems. It is the underlying protocol for data communication on the World Wide Web. The implementation of HTTP follows the principles laid out in REST architectural style and becomes a natural choice of transport protocol for any APIs that need to be built for accessibility on the Web.

The figure below shows the components that make up a typical API Management Platform along with the interacting partners. Some prominent components of this figure feature –

1. API Gateway
2. Management and configuration layer including security, monitoring and other services
3. Backend APIs
4. Developer Portal
5. Developer Community
6. User Community



An API Gateway provides the functionality of a reverse proxy that redirects the incoming requests to the appropriate backend services.

The Management layer allows for configuration of various utilities that the platform provides. It allows for configuring security, publishing APIs, adding policies on APIs or package group of APIs, configuring developer portal, configuring analytics on APIs, etc. Some popular platforms allow for data transformation as part of this layer. Most platforms also allow for caching through various database implementation as part of API management.

The developer portal allows for developers to test and validate the available APIs. It provides an onboarding experience for the developers to discover API offerings, test, validate and subscribe as needed. The APIs that are developed as part of the product offerings may be used by the Developer Community that package it as part of their wider array of offerings within their applications. One of the critical factors that drives effective API design is to ensure they are intuitive to use for application developers. The easier and intuitive they are, the more developers embrace your APIs. In this paper, our goal is to ensure that the various components or entities that make up an API are broken down and discussed for design.

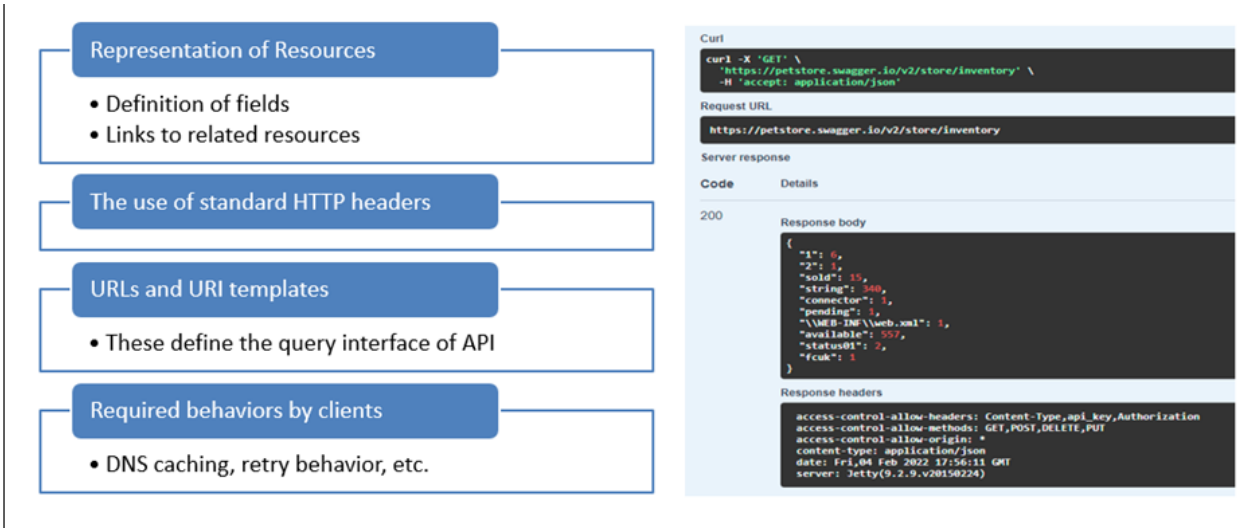
The topic of API management can be fairly large that would need an entire course of topics including API Design, Governance, Implementation, Lifecycle management, etc. In this paper, we shall discuss on individual components that should be considered for API design.

API Design Elements

An API design may be composed of the following components

1. Representation of Resources
2. Use of HTTP Headers
3. URLs and URI templates
4. Required client behaviour

The illustration below shows the various components as part of an API Request/ Response.



In this paper, we shall discuss the design considerations with regards to the object representations of the resources, the usage of HTTP headers and how they can improve developer experience. We shall also cover the URL/URI templates that can be used.

Data Oriented Approach

HTTP has a set of standard operations that make it intuitive for developers to implement CRUD Operations, namely, POST, GET, PUT, DELETE. These operations when combined with the resources on which they are implemented, provide fairly good amount of information to application developers on the purpose of each service.

The resource could be one of the many entities that are part of the system, it could be, for example, a “student” or a “department” in a system that is composed of operations on university resources. When a URL such as the one below points to a certain resource, or in other words, refers to "data", then you are essentially using a data-oriented approach to build your URL.

- **https://www.apiuniversity.com/students/{student_id}**

With the URL that refers to a resource as shown above, it becomes easy to understand, by nature of its representation, that the appropriate HTTP Operations on this resource that may be used for corresponding CRUD operations.

Operation (CRUD)	HTTP Operation	URL
Create a new resource	POST	https://www.apiuniversity.com/students/{student_id}
Read a resource	GET	https://www.apiuniversity.com/students/{student_id}
Update a resource	PUT	https://www.apiuniversity.com/students/{student_id}
Delete a resource	DELETE	https://www.apiuniversity.com/students/{student_id}

When compared with naming through function-oriented approach, as shown in the URL below,

- **https://www.apiuniversity.com/getStudent?id={student_id}**

it becomes apparent that the designer uses his discretion or a certain standard that duplicates the information that is already provided through the HTTP Operation. Also, it is quite possible that the API user may be integrating several systems of which the university services is one of them, and with a function-oriented approach, he has a learning curve to go through to gain familiarity on the functions used to represent appropriate operations.

Through data-oriented approach, a uniformity on the representations is achieved that helps application developers that are using your APIs.

Designing Representations

Data exchange on the web has seen the usage of many formats - EDI, XML, JSON to name a few. Each format has its own advantages and limitations. For example, EDI can pack a lot more into bytes while XML can be more descriptive. Both formats are highly mature with their own use-cases. JSON is derived from JavaScript and highly adopted in APIs for the simplicity of its representation and human-readability. JSON has well-understood parsers and is light-weight. It is currently the dominant media type for APIs and well-understood by the user community.

Each resource within a system may be represented through a JSON object and invoking HTTP GET operation on the resource may return it. Along with the properties of the resource, it is recommended to include a link with a URL that points back to the resource, also called a self-link. A self-link may not seem particularly useful at first thought, but from an API consumer point of view, it provides a context on the returned object. It is also possible that the implementation unit that receives the response at the API consumer end is not necessarily the same one that invoked the request, and in such cases, it is valuable to provide a context to the returned object.

An object may have self-links and contain links to other related resources or resource groups. For instance, in a university system, an HTTP GET operation on a “department” may provide a self-link and along with it may also include a link for students, staff in the department. See illustration below.

```
{
    "self": "https://www.university.com/departments/D789",
    "kind": "department",
    "name": "Electrical Engineering",
    "id": "D789",
    "location": "123 Main Street, Univ City, 1D3FC",
    "students": "https://www.university.com/departments/D789/students",
    "staff": "https://www.university.com/departments/D789/staff"
}
```

It is also worth noting that the links may not necessarily be absolute URLs as shown in the illustration above. Using a relative path helps with easier translation between multiple environments but comes with an overhead of conversion to the absolute URLs.

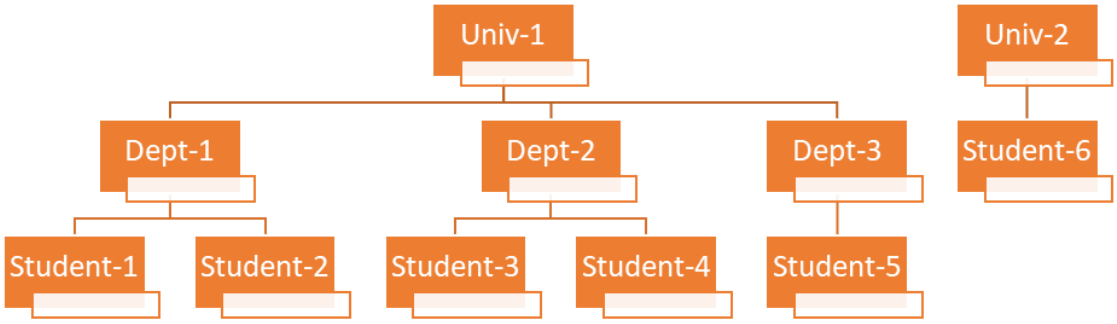
Designing Entity URLs

Designing URLs is more than just keeping it unique. For one thing, it needs to be simple and reproduceable. When the components of a URL are well-defined and easy to understand, it is makes it easy for API developers and consumers to parse and reconstruct as needed. Compare the below URLs that drive home the point:

- https://www.apiuniversity.com/students/S123
- https://www.apiuniversity.com/students/0lkjeNhu123Nthy;124nhysgAS0098nC

In the first example, it was intuitive that a student id was used as part of the URL. In the second case, the URL could have been produced using a session id.

One of the blind spots during API design may be embedding of hierarchies into a URL. This situation may be best illustrated through an example.



One could design the student resource URL in one of the following ways to refer to a student:

- https://www.apiuniversity1.com/depts/D0001/students/S0001
- https://www.apiuniversity2.com/students/S0006

In the first example above, it is necessary that a student department and the id are available to be able to retrieve student information, while in the second case, student information may be retrieved with just the student id. Furthermore, if the "student" object is designed well, one could as well find the department he belongs to through the properties and (/or) links available as part of student object representation. Also, it is not uncommon for business use-cases to emerge in a system that may not fit the hierarchy anymore.

This is not to say that hierarchies are irrelevant, but one should be aware of the blind spots that one may run into when it comes to designing the URL.

Using Property: "kind"

An object representation may include a property “kind” to represent the object type. Some advantages are obvious, the “kind” conveys to the API consumer about the object type that is in play and helps decide the parser to use for further processing. However, in most situations, the object type in the response may be predictable. For example, it may seem that when an HTTP GET operation on a /student resource is invoked the returned object is always of “kind”: “student”. But let us investigate this further with a use case. Consider an upgrade to the existing implementation of the students API. The newer version is supposed to retrieve student information for students that joined through exchange program as well. These students may have extra properties or a distinct set of properties than the original object representation of student. Addition of the properties to the same object may result in unexpected outcomes on the API consumer code that is already coded for the older version. Instead, let us define a new object, for example, “kind”: “exchange-student” that may be returned as part of the student API. With the usage of “kind” property, the API consumer would now have a better control in deciding which ones to process and the ones to ignore. This strategy would help in avoiding runtime issues due to changes.

As can be observed from above, some advantages are obvious, while some that are realized through specific business use cases.

Collections

Any system design deals with several types of resources, and their relationships, and an effective way to represent a collection of resources is essential. The illustration below shows a JSON object containing a JSON array of students. There could be a tendency to omit the outer JSON and represent with just the JSON array, which may make it much simpler. But there are certain advantages of using the parent structure. It helps provide an appropriate place to add properties related to the collection, for example, the last modified date of the collection.

Based on the business need, in some cases, the JSON array may only contain individual object URLs, rather than all properties.

```
{
    “self”: “https://www.student-info.com/students”
    “kind”: “Collection”
    “contents”: [
        {
            “self”: “https://www.student-info.com/students/123”,
            “kind”: “student”,
            "student_id": 123,
            “name”: “John Doe”,
            “department”: “Computer Science”,
            “email”: “john.doe@student-info.edu”,
            “postal_code”: 66502
        },
        {
            “self”: “https://www.student-info.com/students/812”,
            “kind”: “student”,
            "student_id": 812,
            “name”: “Mike Smith”,
            “department”: “Plant Pathology”,
            “email”: “mike.smith@student-info.edu”,
            “postal_code”: 66502
        }
    ]
}
```

Designing Query URLs

The idea of having the ability to filter a result set through a set of parameters is not new. The semantics of invocation and implementation of Query URLs have their origins in RPC method invocation that were in use much earlier than the emergence of APIs and the query parameters. A URL may be loaded with appropriate parameters to filter a larger resource set to narrow down the results.

While implementing a query URL, it is essential to filter result sets, it is recommended that the query parameters are used with care. Let us discuss more based on the two URLs below

- https://www.apiuniversity.com/students/S0001
- https://www.apiuniversity.com/students?id=S0001

Both examples above use noun-based URLs, but the way student id is passed on to the application is different. A general recommendation is to use the primary key of your resource as part of the URL, as shown in the first example above. Usage of parameters can be implemented based on the need. It turns out that API developers find it easy to implement as well. Using this format also helps build URLs that express graph traversals. For example, a URL such as the one below, may be used to retrieve the list of staff from a department that student with id, S0001 belongs to. In the next section, we shall investigate the pattern that emerges out of representing relationships in this way.

- https://www.apiuniversity.com/students/S0001/department/staff

Representing Relationships

Building on the query URL that we presented with regards to the graph traversals in the previous section, a certain pattern may be spotted. Relationships are expressed through the following expressions
/{entity-group}/[{entity-id}]/…/{entity-group}/[{entity-id}]

An entity-id may be used when there is a multi-valued relationship from the previous segment.

It is also possible that relationships could be many-to-many, and hence the URLs can be symmetric in nature, for example, a student may enroll into multiple courses, while each course may be enrolled by multiple students. Listed below are examples for this scenario.

- https://www.apiuniversity.com/students/S0001/courses
- https://www.apiuniversity.com/courses/C7001/students

If there are query needs that are beyond the traversal of relationships, query string parameters may be used to accomplish the same, as shown in the example below.

- https://www.apiuniversity.com/departments/D7801/courses?semester=spring

Representing Non-persistent Resources

There are some scenarios when the target of an API is not a persistent resource. The target could be certain function that the system provides. An enterprise may provide functionality for a simple math function or a complex function such as calculation of income tax owed. An API could also be utilized for unit conversions such as metrics units to SI, for example, or translation of a language. In such scenarios the target is a function loaded with certain parameters.

When designing for such systems, it is still possible to use noun-based URLs such as the below examples.

- https://www.apiuniversity.com/tuition?department=D0981&semester=5
- https://www.apiuniversity.com/tuition/D0981/5

Usage of noun-based URLs, rather than verbs such as /getTuition, help with maintaining the uniformity of URLs for persistent and non-persistent resources across the system.

In some situations, if there are more than a manageable number of variables to code for, especially with complex functions, an API request body may be used with a JSON object as shown below

HTTP:/tuition

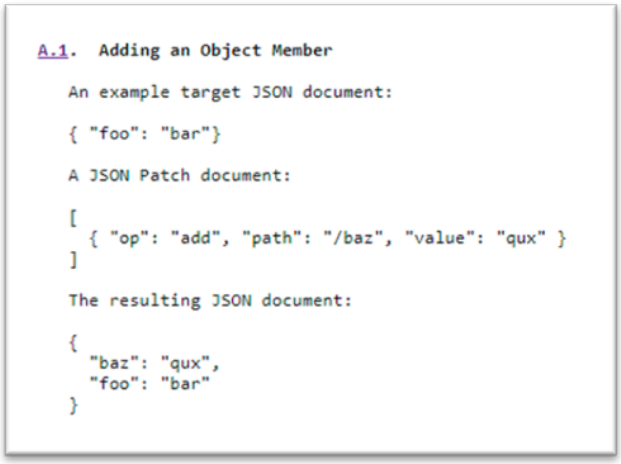
Body

```
{
    “department”: “D0001”
    “semester”: 5
}
```

Supporting HTTP PATCH

In a typical system, a resource may be updated multiple times, and more often than not, the update may be needed only on a subset of properties. When a PUT operation is used, the entire content of resource is modified, and the onus of preserving the properties, other than the ones that need to be updated, is with the client.

Furthermore, if there is a process in the system that is configured to capture updates to a certain property of a resource, this could trigger jobs that may not be needed in the first place. There is a loss of processing power and overhead of preserving the properties. In such scenarios, HTTP Patch provides a great way to update only the properties that need updates on a resource.



A.1. Adding an Object Member

An example target JSON document:

```
{ "foo": "bar"}
```

A JSON Patch document:

```
[
  { "op": "add", "path": "/baz", "value": "qux" }
]
```

The resulting JSON document:

```
{
  "baz": "qux",
  "foo": "bar"
}
```

Provided here is an illustration for adding a property info a resource, taken from https://datatracker.ietf.org/doc/html/rfc6902, the JSON Patch Standards from Internet Engineering Task Force (IETF). The link provides implementation for other operations as well.

Pagination & Partial Response

For Pagination, URL with query parameters for limit and offset may be used. Offset may provide the start record and “limit” may indicate the number of records that are requested from the start record. A series of such URLs may be used from the Application to fetch the next or previous records as needed. Usage of pagination is encouraged for resources that have lot of records and may need large bandwidths to load all data. An example URI shown below

- /students?limit=100,offset=300

Another way of saving bandwidth may be to implement a partial response feature. In this strategy, a query parameter such as “fields” may be used to indicate the properties that are requested as part of the response. If a resource object representation contains a lot of properties, and if the result set is huge, then a narrow-down approach to the response payload helps with avoiding the unnecessary data transfer and also keeps the application code simple. Example for such a URI is shown below

- /students?fields=student_id,name,dept

Response Codes & Authentication

HTTP is ubiquitous on the internet; it is one of the most well understood and implemented protocol for data exchanges on the web. Its usage over the decades makes it a hardened and mature protocol. HTTP comes with a standard set of response codes for successful and error scenarios. Usage of the standard codes in implementation makes it easy for application developers to get a head-start on the processing of responses in the application code, rather than re-learning custom codes specific to the system.

Response codes also have standard header properties that may be used along with the code. For example, along with response code “201 Created”, provide the location header with the URL of the newly created header. Similarly, successful response to a GET operation may accompany relevant header data such as Content-Type, Content-Location etc. For an error response such as 405 error code for unsupported method, map the header property “Allow” with the supported methods.

For authentication, it is encouraged to use OAuth2. It is highly effective as it allows the API Provider to grant and revoke access remotely. It also allows for easy distinction of the specific party that is invoking requests and helps with tracking.

Complementing with an SDK

A well-designed API that is standards-based, and well-documented, should be just enough for application developers to consume, ideally. One of the quick-start items that enhances developer experience is a collection of ready-to-go samples in the language of their choice. These artifacts help developers get started with minimum effort. An onboarding experience for a consumer may include some code samples in the language of their choice.

However, most developers have a preference of using an SDK in the language of their choice. To get started on catering to such needs, an analysis of the audience that is using your APIs becomes a primary driving factor in deciding the language-choices that you want to build SDKs with. Most developers may be treating the SDK as something that is independent of the API, while in reality, it is just another wrapper that is calling the API.

Also, of note, is that with a well-designed API, it is relatively straight-forward to create an SDK out of it. It is usually a wrapper with the same functionality that is invoked from the programming language of choice.

Conclusion

One of the most significant gains that we have seen in sticking to HTTP and URI specifications is that it minimizes the amount of learning needed for a consumer. The entity-oriented approach requires programmers/consumers to build a mapping between API model and their code, but it allows for simplicity and uniformity of APIs. It also helps maintain granularity when one decides to build an SDK. This granularity level also allows developers to integrate with the needed resources based on the business needs, rather than working with multiple aggregated resource models. This entity-level granularity also helps build an SDK for a specific programming language.

It is recommended to consider links as part of the object representation, as there is a growing appreciation for the value of links in APIs. As discussed in the earlier sections, links help provide the context for the implementation units that process the response.

The property ‘kind’ helps identify the object returned by the API response, and the processing unit may decide its workflow based on the ‘kind’ of the object. It is recommended to treat collections as just another entity representation with its own set of properties along with the child objects of the set. The child objects may be represented in full or through links if the business use case allows for it; usage of links helps reduce bandwidth needs. We have also discussed that non-persistent entities may still be represented through noun-based URLs and the ways in which they can be designed.

For completeness, it is recommended to design for HTTP Patch operation. Using a Patch operation helps retain the resource properties that do not need to be updated. We discussed how the onus on the developers is different when using HTTP Put vs HTTP Patch. As part of the exception handling, it is recommended that the existing HTTP response codes be used for uniformity and familiarity. Finally, consider building a great developer experience with code snippets for various programming languages for a quick start of implementation. If an SDK is considered, start with analyzing the audience and narrow down on the programming languages to consider.

The pace at which APIs are being adopted by enterprises and embraced by the developer communities, it makes an essential part of any enterprise strategy. The adoption also fosters much innovation in today’s digital economy; use cases hitherto haven’t been thought of are now handled through seamless integration, thanks to APIs. A huge part of how applications could integrate across so many diverse systems is due to APIs. The fintech industry is a case in point. Many sectors, including travel, logistics, retail, consumer goods, healthcare, etc. that we work with everyday have their digital strategy built around APIs for B2B and D2C interactions.