1.Parameters (API reference tutorial)

Parameters are options you can pass with the endpoint (such as specifying the response format or the amount returned) to influence the response. There are several types of parameters: header parameters, path parameters, and query string parameters. Request bodies are closely similar to parameters but are not technically a parameter.

The different types of parameters are often documented in separate groups on the same page. Not all endpoints contain each type of parameter.

2. Create an API that would save a new entry with all the relevant properties which retrieves values from the endpoint GET /entries.

REST APIs are used to access and manipulate data using a common set of stateless operations. These operations are integral to the HTTP protocol and represent essential create, read, update, and delete (CRUD) functionality, although not in a clean one-to-one manner:

- `POST` (create a resource or generally provide data)
- `GET` (retrieve an index of resources or an individual resource)
- `PUT` (create or replace a resource)
- `PATCH` (update/modify a resource)
- `DELETE` (remove a resource)

Using these HTTP operations and a resource name as an address, we can build a Node.js REST API by creating an endpoint for each operation. And by implementing the pattern, we will have a stable and easily understandable foundation enabling us to evolve the code rapidly and maintain it afterward. The same foundation will be used to integrate third-party features, most of which likewise use REST APIs, making such integration faster.

For now, let's start creating our secure Node.js REST API.

In this tutorial, we are going to create a pretty common (and very practical) secure REST API for a resource called `users`.

Our resource will have the following basic structure:

- `id` (an auto-generated UUID)

- `firstName`
- `lastName`
- `email`
- `password`
- `permissionLevel` (what is this user allowed to do?)

And we will create the following operations for that resource:

- `POST` on the endpoint `/users` (create a new user)
- `GET` on the endpoint `/users` (list all users)
- `GET` on the endpoint `/users/:userId` (get a specific user)
- `PATCH` on the endpoint `/users/:userId` (update the data for a specific user)
- `DELETE` on the endpoint `/users/:userId` (remove a specific user)

3) Question: what are the key things you would consider when creating/consuming an API to ensure that it is secure and reliable?

Ans;

Understanding the Potential Risks of APIs

The downside of publicly available web APIs is that they can potentially pose great risk to API providers. By design, APIs give outsiders access to your data: behind every API, there is an *endpoint*—the server (and its supporting databases) that responds to API requests (see Figure 1). In terms of potential vulnerability*A vulnerability is an inherent weakness in a system (hardware or software) that an attacker can potentially exploit. Vulnerabilities exist in every system; "zero-day" vulnerabilities are those that have not yet been discovered.*, an API endpoint is similar to any Internet-facing web server; the more free and open access the public has to a resource, the greater the potential threat from malicious actors. The difference is that many websites at least employ some type of access control, requiring authorized users to log in. One problem with some APIs, as we'll see shortly, is that they provide weak access control and, in some cases, none at all. With APIs becoming foundational to modern app development, the attack surface is continually increasing. Gartner estimates that "by 2022, API abuses will move from infrequent to the most frequent attack vector resulting in data breaches for enterprise web applications.