Identify REST contraints when consuming APIs

REST Constraints

REST defines six architectural constraints that make any web service a truly RESTful API. These are constraints also known as Fielding's constraints (see https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). They generalize the web's architectural principles and represent them as a framework of constraints or an architectural style. These are the REST constraints:

- Client/server
- Stateless
- Cache
- Uniform interface
- Layered system
- Code on demand

Client/Server

The client and server exist independently. They must have no dependency of any sort on each other. The only information needed is for the client to know the resource URIs on the server. The interaction between them is only in the form of requests initiated by the client and responses that the server sends to the client in response to requests. The client/server constraint encourages separation of concerns between the client and the server and allows them to evolve independently.

Stateless

REST services must be stateless. Each individual request contains all the information the server needs to perform the request and return a response, regardless of other requests made by the same API user. The server should not need any additional information from previous requests to fulfill the current request. The URI identifies the resource, and the body contains the state of the resource. A stateless service is easy to scale horizontally, allowing additional servers to be added or removed as necessary without worry about routing subsequent requests to the same server. The servers can be further load balanced as necessary.

Cache

With REST services, response data must be implicitly or explicitly labeled as cacheable or non-cacheable. The service indicates the duration for which the response is valid. Caching helps improve performance on the client side and scalability on the server side. If the client

has access to a valid cached response for a given request, it avoids repeating the same request. Instead, it uses its cached copy. This helps alleviate some of the server's work and thus contributes to scalability and performance.

Note

- GET requests should be cacheable by default. Usually browsers treat all GET requests as cacheable.
- POST requests are not cacheable by default but can be made cacheable by adding either an Expires header or a Cache-Control header to the response.
- PUT and DELETE are not cacheable at all.

Uniform Interface

The uniform interface is a contract for communication between a client and a server. It is achieved through four subconstraints:

- **Identification of resources**: Resources are uniquely identified by URIs. These identifiers are stable and do not change across interactions, even when the resource state changes.

- **Manipulation of resources through representations**: A client manipulates resources by sending new representations of the resource to the service. The server controls the resource representation and can accept or reject the new resource representation sent by the client.

- **Self-descriptive messages**: REST request and response messages contain all information needed for the service and the client to interpret the message and handle it appropriately. The messages are quite verbose and include the method, the protocol used, and the content type. This enables each message to be independent.

- **Hypermedia as the Engine of Application State (HATEOS)**: Hypermedia connects resources to each other and describes their capabilities in machine-readable ways. Hypermedia refers to the hyperlinks, or simply links, that the server can include in the response. Hypermedia is a way for a server to tell a client what HTTP requests the client might want to make in the future.

Layered System

A layered system further builds on the concept of client/server architecture. A layered system indicates that there can be more components than just the client and the server,

and each system can have additional layers in it. These layers should be easy to add, remove, or change. Proxies, load balancers, and so on are examples of additional layers.

Code on Demand

Code on demand is an optional constraint that gives the client flexibility by allowing it to download code. The client can request code from the server, and then the response from the server will contain some code, usually in the form of a script, when the response is in HTML format. The client can then execute that code.

REST API Versioning

Versioning is a crucial part of API design. It gives developers the ability to improve an API without breaking the client's applications when new updates are rolled out. Four strategies are commonly employed with API versioning:

- **URI path versioning**: In this strategy, the version number of the API is included in the URL path.
- **Query parameter versioning**: In this strategy, the version number is sent as a query parameter in the URL.
- **Custom headers**: REST APIs are versioned by providing custom headers with the version number included as an attribute. The main difference between this approach and the two previous ones is that it doesn't clutter the URI with versioning information.
- **Content negotiation**: This strategy allows you to version a single resource representation instead of versioning an entire API, which means it gives you more granular control over versioning. Another advantage of this approach is that it doesn't require you to implement URI routing rules, which are introduced by versioning through the URI path.

Pagination

When a request is made to get a list, it is almost never a good idea to return all resources at once. This is where a pagination mechanism comes into play. There are two popular approaches to pagination:

- Offset-based pagination
- Keyset-based pagination, also known as continuation token or cursor pagination (recommended)

A simple approach to offset-based pagination is to use the parameters offset and limit, which are well known from databases.

Below shows how query parameters are passed in the URI in order to get data based on offset and to limit the number of results returned.

```
http://myhouse.cisco.com/api/room/livingroom/devices?offset=100&limit=15 # returns the devices between 100-115
```

Default values are used if the parameters are not specified.

```
http://myhouse.cisco.com/api/room/livingroom/devices # returns the devices 0 to 200
```

Note that the data returned by the service usually has links to the next and the previous pages, as shown below:

```
GET /devices?offset=100&limit=10
{
  "pagination": {
    "offset": 100,
    "limit": 10,
    "total": 220,
  },
  "device": [
    //...
  ],
  "links": {
    "next": "http://myhouse.cisco.com/devices?offset=110&limit=10",
    "prev": "http://myhouse.cisco.com/devices?offset=90&limit=10"
  }
}
```

Rate Limiting and Monetization

Rate-limiting techniques are used to increase security, business impact, and efficiency across the board or end to end.

Security: unlimited access to API could have negative effects such as decrease value and limit business success. Rate limiting is a critical component of an API's scalability. Processing limits are typically measured in transactions per second (TPS). If a user sends too many requests, API rate limiting can throttle client connections instead of disconnecting them immediately. Throttling enables clients to keep using your services while still protecting your API. Risk of API requests timing out, and the open connections also increase the risk of DDoS attacks.

Business impact: One approach to API rate limiting is to offer a free tier and a premium tier, with different limits for each tier. Limits could be in terms of sessions or in terms of number of APIs per day or per month. API providers need to consider the following when setting up API rate limits:

- Are requests throttled when they exceed the limit?
- Do new calls and requests incur additional fees?

- Do new calls and requests receive a particular error code and, if so, which one?

Efficiency: Unregulated API requests usually and eventually lead to slow page load times for websites.

Rate Limiting on the Client Side

Various rate-limiting factors can be deployed on the server side. As a good programming practice, if you are writing client-side code, you should consider the following:

- Avoid constant polling by using webhooks to trigger updates.
- Cache your own data when you need to store specialized values or rapidly review very large data sets.
- Query with special filters to avoid re-polling unmodified data.
- Download data during off-peak hours.