How to build an API

Diogo Gomes dgomes@ua.pt

How to build an API (1)

- Make sure that the base URL of the API is simple.
  - E.g.: if we want to design APIs for products:

```
/products
/products/12345
```

  - The first API is to get all products and the second one is to get specific product.
- **Use nouns and NOT the verbs**
- A lot of developers make this mistake. They generally forget that we have HTTP methods with us to describe the APIs better and end up using verbs in the API URLs.
  - E.g.: DO **NOT** do:

```
/getAllProducts
```

How to build an API (2)

- **Use of right HTTP methods**
- RESTful APIs have various methods to indicate the type of operation you are going to perform with this API
  - **GET**—To get a resource or collection of resources.
  - **POST**—To create a resource or collection of resources.
  - **PUT/PATCH**—To update the existing resource or collection of resources.
  - **DELETE**—To delete the existing resource or the collection of resources.
- You need to make sure we use the right HTTP method for given operation.

How to build an API (3)

- **Use Plurals**

  `/products`

- plural avoids confusion whether we are talking about getting single resource or a collection. It also avoids adding additional things like attaching all to the base URL
  - E.g. /product/all

How to build an API (4)

- **Use parameters**
- Sometimes you might need to have an API which should provide more data than just by id. You should make use of query parameters to design the API.
  - `/products?name='ABC'` should be preferred over `/getProductsByName`
  - `/products?type='xyz'` should be preferred over `/getProductsByType`
- This way you can avoid long URLs with simplicity in design.

How to build an API (5)

- **Use proper HTTP codes**
- We have plenty of HTTP codes. Most of us only end up using two—200 and 500! This is certainly not a good practice. Following are some commonly used HTTP codes.
  - **200 OK**—This is most commonly used HTTP code to show that the operation performed is successful.
  - **201 CREATED**—This can be used when you use POST method to create a new resource.
  - **202 ACCEPTED**—This can be used to acknowledge the request sent to the server.
  - **400 BAD REQUEST**—This can be used when client side input validation fails.
  - **401 UNAUTHORIZED / 403 FORBIDDEN**— This can be used if the user or the system is not authorised to perform certain operation.
  - **404 NOT FOUND**— This can be used if you are looking for certain resource and it is not available in the system.
  - **500 INTERNAL SERVER ERROR**—This should never be thrown explicitly but might occur if the system fails.
  - **502 BAD GATEWAY**—This can be used if server received an invalid response from the upstream server.

How to build an API (6)

- **Versioning**
- Versioning of APIs is very important. Many different companies use versions in different ways, some use versions as dates while some use versions as query parameters. Best practice is nonetheless to include it in the URL. E.g.

```
/v1/products
/v2/products
```

- Avoid using /v1.2/products as it implies the API would be frequently changing. Also dots (.) might not be easily visible in the URLs. So keep it simple.
- It is always good practice to keep backward compatibility so that if you change the API version, consumers get enough time to move to the next version.

How to build an API (7)

- **Use Pagination**
- Use of pagination is a must when you expose an API which might return huge amounts of data and if proper load balancing is not done, then the consumer might end up bringing down the service.
- We need to always keep in mind that the API design should be full proof and fool proof.
- Use of limit and offset is recommended here. For example, `/products?limit=25&offset=50` It is also advised to keep a default limit and default offset.

How to build an API (8)

- **Supported Formats**
- If is also important to choose how your API responds. Most of the modern day applications should return JSON responses unless you have an legacy app which still needs to get XML response.

How to build an API (9)

- **Use Proper Error Messages**
- It is always a good practice to keep a set of the error messages application sends and respond that with proper id. For example, if you use Facebook graph APIs, in case of errors, it returns message like this

-
  ```
  {
      "error": {
          "message": "(#803) Some of the aliases you requested
  do not exist: products",
          "type": "OAuthException",
          "code": 803,
          "fbtrace_id": "FOXX2AhLh80"
      }
  }
  ```
- It's a good practice to return an URL with an error message which tells about the error message and how to handle it as well.

How to build an API (10)

- **Use of Open API specifications**
  - In order to keep all teams in your company abide to certain principles, use of OpenAPI Specification can be useful. Open API allows you to design your APIs first and share that with the consumers in easier manner.

GraphQL

- released by Facebook publicly in 2015
- GraphQL is a query language, specification, and collection of tools, designed to operate over a single endpoint via HTTP, optimizing for performance and flexibility.
- GraphQL APIs are organized in terms of types and fields, not endpoints.
- GraphQL uses types to ensure Apps only ask for what's possible and provide clear and helpful errors.
  GraphQL allows to generate a lot less network calls.
-

More info: https://graphql.org/learn/

Core Principles

- GraphQL exposes a single endpoint.
- You send a query to that endpoint by using a special Query Language syntax.
- The server responds to a query by providing a JSON object.
- GraphQL has its own type system that's used to define the schema of an API. The syntax for writing schemas is called (SDL).

- You can plugin GraphQL into your projects: https://graphql.org/code/

Example GraphQL query

Definition

```
type Query {
    me: User
}
type User {
    id: ID
    name: String
}
```

Request

```
query {
    me {
        name
    }
}
```

JSON result

```
{
    "me": {
        "name": "Luke Skywalker"
    }
}
```

Example GraphQL query with arguments

Request

```
{
  human(id: "1000") {
    name
    height
  }
}
```

JSON Result

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 1.72
    }
  }
}
```

GraphQL operation

Queries (read)

```
query {
    search(q: “name”) {
        title
        author
    }
}
```

Mutations (write)

```
mutation {
    create(title: “book”) {
        id
    }
}
```

Subscriptions

```
subscription {
    onCreate {
        id
        title
    }
}
```

Comparison REST vs GraphQL

	REST	GraphQL
Data Fetching	Gathers data by accessing multiple endpoints	Single Query to the GraphQL server that includes the concrete data requirements
Over- and Underfetching of data	Often over- and underfetches data	Exposes all of the data from a single endpoint
Rapid Product Iterations on the Frontend	Every change in the frontend can lead to more/less data required in the API	Frontend always gets what is asks for, with maximum flexibility on the backend
Type System	N/A	GraphQL Schema Definition Language (SDL)
Analytics	Over and Underfetching limits analysis	Fine-grained insights about the data requested

Websockets

- IETF Protocol and W3C JavaScript API for real-time, bi-directional, always-on connections
- RFC 6465 (Dec 2011)
- Easily add event-based data to web apps
- Avoids polling
- Avoids HTTP meta-data overhead
- Traverses Firewalls

- Cons: You must the define an Application-Level Protocol
  - You can re-use other protocols: JMS, XMPP, MQTT, etc