Design of a Good API

Tips and guideline on the design of APIs

Ayoub Ouarrak

13/03/2019

General Principles

KISS (keep it simple stupid)

Anyone should be able to use your API whithout having to referer to a documentation, in general Characteristics for good API are:

- Easy to learn
- Easy to use (even without documentation)
- Easy to extend
- ...

SRP: Single Responsibility Principle

An API should do one thing and do it well.
The same principle should be applied to Classes/Modules/Microservices.

- Avoid heavy-weight, bloated, or "god" classes
- If a class does everything, it is probably a bad design

Getting this right, will make the API more robust and make modularity less rigid.

Minimize Accessibility and Immutability

- Classes and memebers should be as private as possible
- Public classes should not have public fields (with exception of constants)
- This maximizes **information hiding**

Unless there is a good reason, variables, classes, and collections should be immutable.

Names matters

Avoid naming classes, packages, or libraries anything too generic, such as "ExtrasLibrary". This encourages dumping unrelated code there.

- Names should be self-explanatory
- Consistency, same word means same thing
- Code should read like prose

Class and Method Design

Subclassing

Subclassing only where it makes sens.

- Subclass only when **is-a** relationship exists
- Otherwise, use composition
- Public classes should not subclass other public classes for ease of implementation

Parameters and return types

- Favor interfaces over classes for input (provide flexibility)
- Use most specific possible input parameter type (Moves error from runtime to compile time, e.g. Collection -> Set/List)
- Don't use string if a better type exists (e.g. string for constants)
- Avoid Long Parameter Lists for methods/constructors
- Use Optional or empty collection instead of null

Fail Fast

Report Errors as Soon as Possible After They Occur

- Compile time is best - static typing, generics
- Avoid the usage of raw types when possible

REST API Design

Identify Object Model

The first step in designing a REST API is identifying the objects which will be presented as resources. e.g.

- Users
- Settings
- ...

Routes design

Use nouns not verbs.
REST API's should be designed for Resources, which can be entities or services, etc., therefore they must always be nouns. e.g.

- */stars* instead of */GetAllStars*
- */stars/:bellatrix*

Endpoints name should not be CamelCase, dash separation is prefered. e.g *../GiantStars/..* should be *../giant-stars/..*

Filter and Sort

Don't create different URIs for fetching resources with filtering, searching, or sorting parameters.

Filtering

Use query parameters defined in URL for filtering a resource:

- */costellations/:orion/stars?name=rigel*

Sorting

ASC and DESC sorting parameters can be passed in URL such as:

- */costellations/:orion/stars?sort=magnitude*

HTTP Verbs

Let the HTTP Verb Define the Action.
API's should only provide nouns for resources and let the HTTP verbs (GET, POST, PUT, DELETE) define the action to be performed on a resource. e.g.

- **GET** */users* = Return all users
- **POST** */users* = Create a new user
- **GET** */users/4501* = Get user with id 4501
- **PUT** */users/4501* = Update user with id 4501
- **DELETE** */users/4501* = Delete user with id 4501

HTTP Status Codes

"The worst thing your API could do is return an error response with a 200 OK status code."".
Return a meaningful status code that correctly describes the type of error, and the choice should be consistent.

2xx Success

- **200** All good, dude
- **201** I created the resource, dude
- **204** No content to show, dude ...

HTTP Status Codes

4xx Client Errors

- **400** Bad request (you sending shit, dude)
- **401** Unauthorized (who is there?)
- **403** Forbidden (you can't touch this, du du do do)
- **405** Method Not Allowed ...

5xx Server Errors

- **500** Internal Server Error ...

Versioning

Versioning APIs always helps to ensure backward compatibility of a service while adding new features or updating existing functionality for new clients.
There are different schools of thought to version your API, but most of them fall under two categories below:

- **Headers**
- **URL** e.g. */v1/costellations/...*

Swagger

Swagger is a widely-used tool to document REST APIs that provides a way to explore the use of a specific API, therefore allowing developers to understand the underlying semantic behavior.
It's a declarative way of adding documentation using annotations which further generates a JSON describing APIs and their usage.

Questions?

References

- **How to Design a Good API and Why it Matters**, by *Joshua Bloch*.