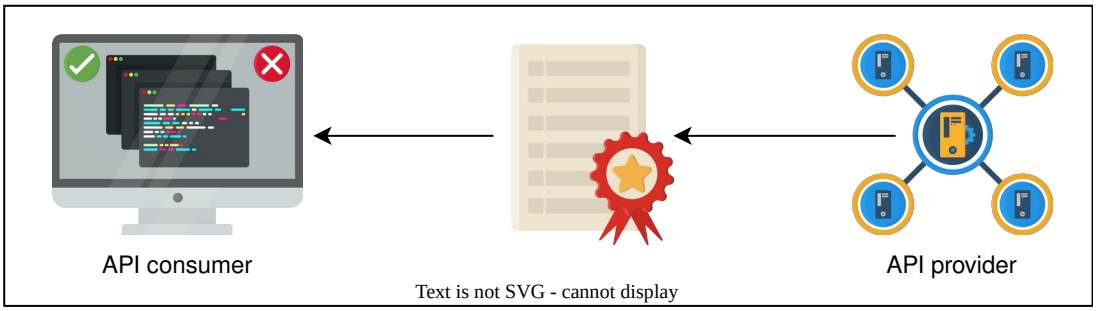
What is API design?

API design is carefully planning, preparing, and developing programming interfaces (APIs) to expose data and the system's functionality to consumers. APIs enable system-to-system communication and are essential for digital organizations because they add new capabilities to their products, operations, partnership strategies, and more. An effective API design is one that has satisfactory answers to the following queries of a developer:

- Why is the API being developed?
- What would be the outcome regarding the impact and output of the system?
- How will the API be designed to meet the requirements?
- What will be the structure of our resources?
- How will we document our resources?

Press+to interact



API serving as a contract between a consumer and provider

Designing an API involves efficiently using remote services to solve problems so that the customer's functional and non-functional needs (more on this later in the course) are satisfactorily met. An API should be well-aligned with the business goals for which it is being developed.

Why does API design matter?

An API is a product similar to a website or a software application. What would be our reaction if we were to interact with a poorly designed product? We'd probably instantly switch to another application with a better design that provides the same functionality. The same thinking applies to APIs as products. It's important to design an API by following all end-user requirements.

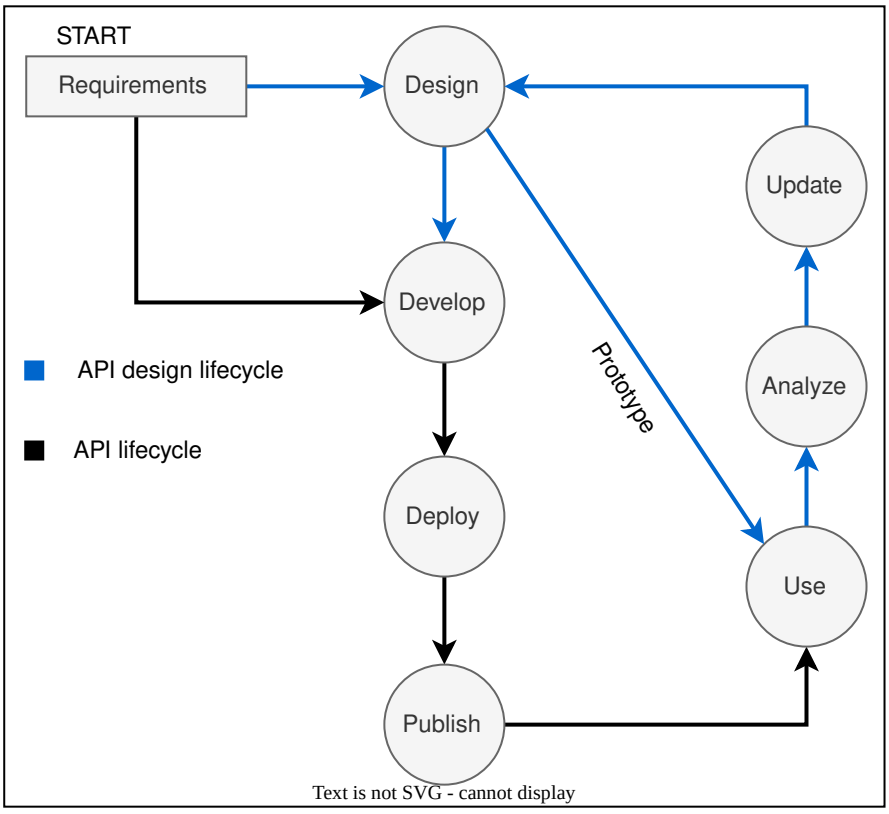
The requirement analysis should be performed before the development of the interface. This analysis is performed during the API design phase. It's possible to update the design of an API by getting internal or customer feedback. Let's see how this happens in the API design lifecycle below.

API design lifecycle

Remember, developers are the customers who are going to use our product (the API). In every product lifecycle, the first phase is to design it according to the functional and non-functional requirements. The same is the case with APIs as well. Like any other product or application,

developers expect APIs to be simple, helpful, and easy to adopt. Therefore, API development follows the same lifecycle as other products, as shown below:

Press+to interact



The API development cycle

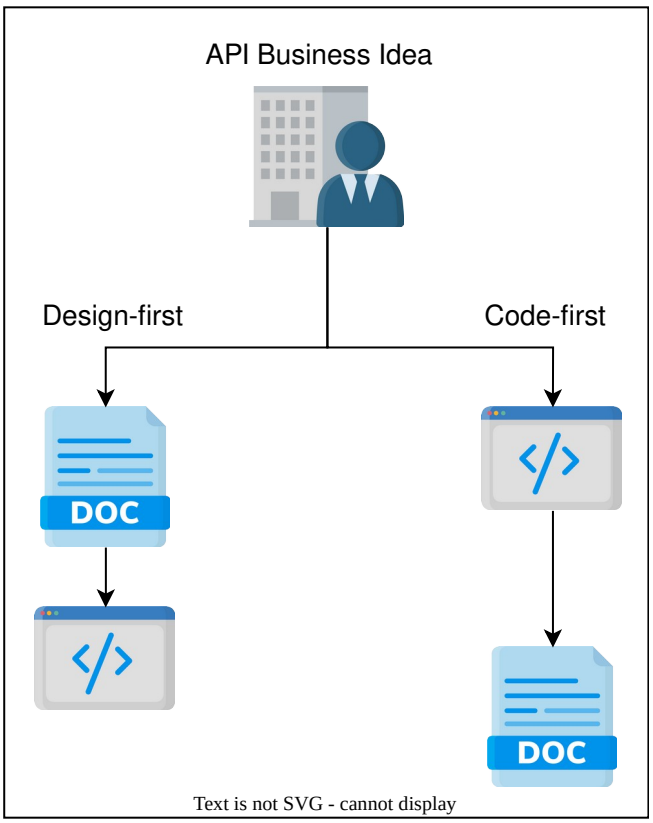
The design-first vs. code-first approach

The **code-first approach** is the traditional way of creating APIs. In this approach, the development of the code starts after the business requirements are laid out, and the documentation is eventually generated from the code. In comparison, the **design-first approach** focuses on creating the API's contract or specification document before writing any code.

The code-first approach is useful in scenarios where we require one (or some) of the following things:

- Quick delivery of the product
- An API for internal use only
- Little or no documentation is required

Press+to interact



The design-first approach vs. the code-first approach

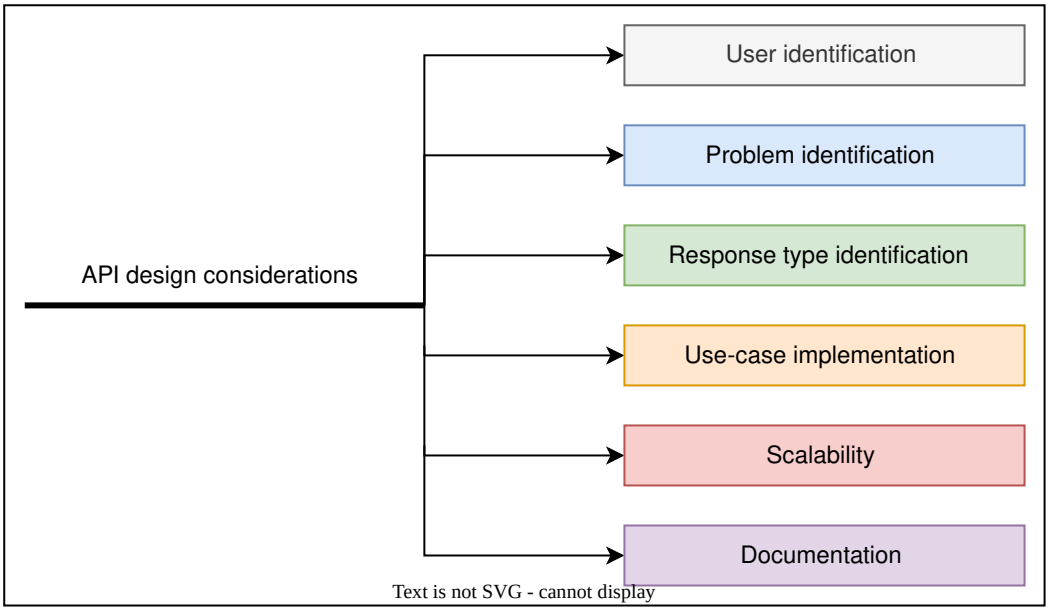
It would make very little sense to begin constructing a house without first having an architect draw up the plans and design of the house. Similarly, it’s clearer for APIs to start with a design-first approach. In most cases, specifications can change depending on the API's end users, which can alter the API's structure. With the design-first approach, we can get early feedback that can save us from future errors, because an API's design can be updated much faster than its code.

> **Note:** The design-first approach allows us to skip the develop, deploy, and publish part of API development. Instead, we can work on a prototype to get feedback and update the design.

API design considerations

The following illustration shows the key points for better API design:

Press+to interact



Key points to consider in API design

- Identify the potential users in terms of their relationship with the business (partners, customers, or external developers) for which API is being developed. This identification will define the level of access and type of authentication to be implemented in the API. The types of users will also help identify and implement the correct architectural style, REST, gRPC, and so on.
- Identify what developer problems are being solved in terms of their business relationship, critical needs, and value addition. Identifying the **metrics** that will improve after using the developed API is also necessary. These metrics include revenue, task speed, cost, and so on.
- It is important to identify the **responses** (successes or errors) to different API calls so that developers can understand the type and reason of the responses received from the server. Exception and error handling must be implemented along with well-defined endpoints of the API.
- Real-life **use-case implementation** should be applied so that the testability factor is checked and developers can understand the effectiveness of the designed API in terms of challenges and opportunities. While designing, we should analyze the performance of the API by testing different possible use cases.
- The API should be **scalable** to handle the increasing demands of customers. Therefore, the API's design should consider scalability as an important factor in order to avoid fundamental modifications in the future.
- The API should be adequately **documented** so the developers can easily understand the integration, behaviors, structures, and parameters to be defined while using the API.

API requirements

We need to define both the functional and non-functional requirements of our APIs. Take a look at both the requirements types below:

- **Functional requirements** define the desired end function of a system and its required parameters. For example, in a video streaming service, the ability to post comments on a video is a functional requirement because it has a defined end goal and parameters.
- **Non-functional requirements** define the performance and quality of the services the API provides. Continuing from the video streaming example above, the ability of an API to quickly respond to the user (low latency) or a number of users posting comments simultaneously (scalability) are forms of non-functional requirements. Other non-functional requirements include availability, reliability, consistency, and so on.

Availability
Scalability
Reliability
Security
DELETE
UPDATE
READ
CREATE
Non-functional
Functional
API requirements
Typical functional and non-functional API requirements

The above diagram shows us some generic requirements. But this list can vary according to the needs of the API.

Characteristics of a good API (design)

There are many desirable characteristics of an API. We've provided a nonexhaustive list of some of them to keep in mind when we study an API or design a new one. However, the evolution of technology often impacts how APIs are specified, and new characteristics can be added in the future.

API Characteristics

Characteristics	Explanation
Separation between API specification and its implementation	• Includes separation between the specification and its implementation, that is, the behavior with the internal structural details • Clean designs allow iterative improvements and changes to the API implementation
Concurrency	• Amount of API calls that can be active simultaneously in a specified period • Useful in ensuring that computing resources are available for all users

Dynamic rate-limiting	• Strategy to limit access to API within a timeframe • Avoids overwhelming the API with an onslaught of requests
Security	• Well-defined security mechanisms for authentication and authorization protocols that will define who can access the API and what parts of the API they can access
Error warnings and handling	• Allows error handling effectively to prevent frustration on the consumer end • Reduces debugging efforts for developers
Architectural styles of an API	• Possible to follow different architectural styles according to its requirements
Minimal but comprehensive and cohesive	• API should be as terse as possible but fulfill its goals
Stateless or state-bearing	• API functions can be stateless and/or maintain their state, but idempotency (operations that yield the same result when they are performed multiple times [Source: Wikipedia]) is a desired feature
User adoption	• APIs that have good adoption often have a devoted user community that helps improve the API over many iterations
Fault tolerance	• Failures are inevitable, but a well-designed API can be made fault-tolerant by using mechanisms that ensure the continued operation of the API, even if some components malfunction
Performance measurement	• There should be appropriate provisions for collecting monitoring data and early warning systems