**What is an API?**

An [**application programming interface**](https://blog.hubspot.com/website/application-programming-interface-api), or API, is the part of an application that communicates with other applications. More technically speaking, an API is a group of protocols and methods that define how two applications share and modify each other’s data.

APIs are necessary in our modern digital infrastructure because they enable standardized and efficient communication between applications which might differ in function and construction.

An API sits between a software’s core components and the public, and external developers can access certain parts of an application’s backend without needing to understand how everything works inside the app. This is what makes an API an *interface* for programmers.

### External vs. Internal APIs

Now, full disclosure: The APIs I’ve talked about thus far are all instances of a specific type of API called a web API. Web APIs facilitate communication between web servers. My examples are also all [**open APIs**](https://blog.hubspot.com/website/free-open-apis), which means that they are available for anyone to use, including third-party software developers.

However, API’s don’t have to be publicly available. Internal APIs are used for communication within an application, and access is usually restricted to the organization’s employees and authorized developers. Knowing this, we can now start to unpack microservices.

## What is a Microservice?

A microservice is a style of software architecture that divides an application’s different functions into smaller components called “services.” When an application is built this way, it’s said to follow a microservice architecture.

(Quick note: Developers often refer to these smaller components as “microservices” themselves. To avoid confusion, I’ll stick to the term “services” when describing these components, and “microservice” when referencing the entire system architecture.)

For example, the microservice architecture for an application like PayNow could consist of individual services for user account management, integrations with online merchants, and user authentication. Each service works like its own small piece of software within the larger system.

Each service within the larger microservices has just one task, but the scope of these tasks is up to the app’s developers. A basic software application may rely on a few services, as is with PayNow. Or, in the case of large software companies, an application might comprise hundreds of granular services with highly specific functions.

### Why Use a Microservice Architecture?

To understand why a microservice architecture is beneficial for software developers, we first need to understand its predecessor, the monolith.

A monolithic application is the opposite of a microservice — instead of assigning different tasks to different self-contained services, every function of the application is handled by a single program.

While it might make sense to start developing an application this way — why create multiple programs to worry about? — adherents to the monolith will encounter issues as their application grows in capability and complexity. Stuffing every aspect of an application into one program makes it difficult to program and release updates, keep track of changes, identify problems, delegate tasks for developers, and generally understand the code.

In other words, everything is so connected inside the monolith that it can be hard to untangle. This created the need for a new type of architecture, hence the rise of microservices. Compared to a monolith, the microservice architecture improves:

* **Updates:** In a microservice application, updating individual services doesn’t require modification of the entire system. This saves time, money, and debugging effort. It also enables rolling updates as opposed to infrequent major updates.
* **Simplicity:** A developer doesn’t have to understand the entire system architecture to understand one aspect of the software.
* **Team organization:** Microservices define boundaries between developer responsibilities. DevOps teams can be assigned to one or more microservices, instead of some portion of a nebulous monolith.
* **Security:** If one service is compromised, it (ideally) won’t affect any other service significantly.
* **Robustness:**Similarly, if one service breaks, other services are unharmed.
* **Flexibility:** If a team wishes to build a service a particular way (e.g. with a different language or framework), they don’t need to worry about how this might impact other components.

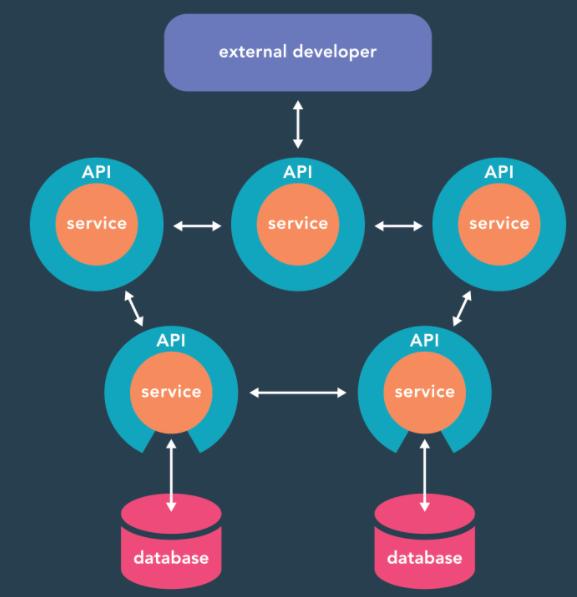
To sum everything up: By separating responsibilities, microservices expedite and simplify the software development process.

Of course, a collection of isolated modules won’t do much good for an application, which is why these services are linked together by — you guessed it — APIs.

## Microservices vs. API

While different things, microservices and APIs are frequently paired together because **services within a microservice use APIs to communicate with each other**. Similar to how an application uses a **public API** to integrate with a different application, one component of a microservice uses a **private API** to access a different component of the same microservice.

Within a microservice, each service has its own API which determines what requests it may receive and how it responds. These APIs typically follow REST principles. Below is a visual example of a basic microservice held together with internal APIs:



Notice how just one module interacts with third-party developers. In our example, this particular service handles integrations with other applications. So, that particular API is **public-facing**, while all other APIs in this microservice are **private**.

It’s important to note that no two microservices are alike, and all utilize APIs differently. Some might assign multiple APIs to one service, or use a single API for accessing multiple services. The visualization above is to help you grasp the overall concept of microservices and APIs, but not every application follows a one-to-one API-to-service pairing.

Finally, remember that APIs have uses beyond microservices. As we discussed, web APIs enable data-sharing between systems, which is necessary for many web applications. Also, APIs can be used internally but without a microservice implementation.

Communication between microservices can be done using:

* REST, GraphQL, gRPC (if we need synchronous communication)
* Kafka, RabbitMQ (if we need asynchronous communication and data which we send must not be lost)