Introduction to Asynchronous APIs

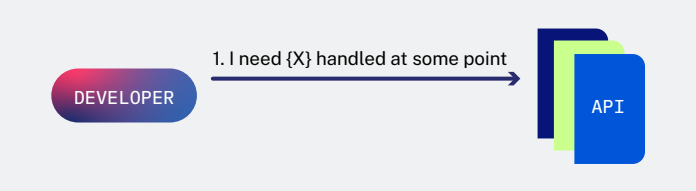
Rapid

Introduction to Asynchronous APIs

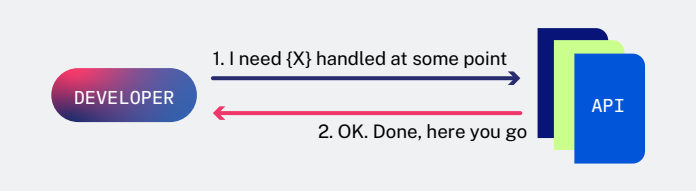
Asynchronous APIs continue to gain traction. In a recent survey of enterprise companies, RapidAPI found the number of developers using async APIs in production nearly quadrupled from 5% in 2019 to 19% in 2020[1]. This document will introduce best practices for using asynchronous APIs, and help you determine if and when to use them.

A Look at Classic (Synchronous) APIs

Before we dig into asynchronous APIs, let's review some characteristics of "classic" API types, including REST and SOAP. These APIs operate in a very transactional way. First, developers make a request to the API.



Then, the request is handled by the API. The developer can expect a near immediate response from the API. These transactions are usually measured in milliseconds, and most will time out after a minute.



Synchronous vs. Asynchronous APIs

With REST and SOAP APIs, the results of the API request are fast and easy to understand. These traits are often desirable in software development, which is why this model is so popular. However, there are some cases where a new model might be helpful. For example:

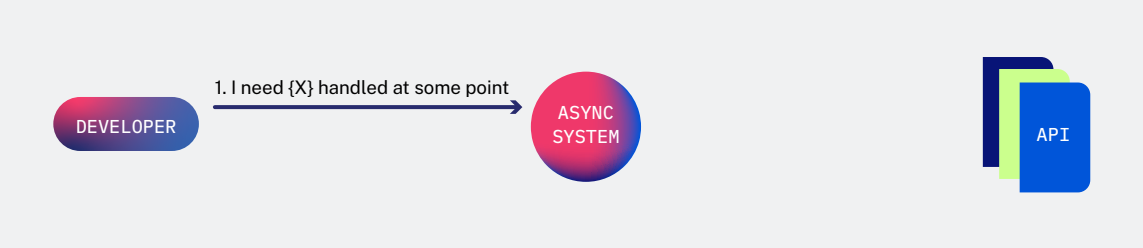
- What if the task is large, or takes a while to complete?
- What if the service is not able to handle the flow of events?
- What if you don't want to be the one to initiate the transaction?

If you are running into these problems, asynchronous APIs can provide a solution. As the name implies, asynchronous APIs do not have to handle requests at the same time the request is made. This is quite different from the classic model used by REST and SOAP APIs.

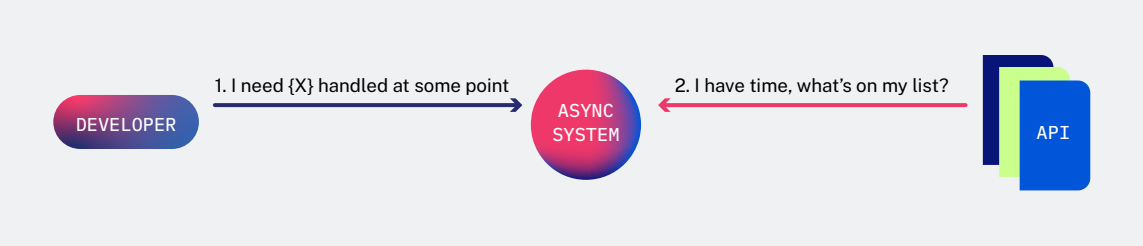
1 RapidAPI Annual Developer Survey 2020-2021

Four Components of Asynchronous APIs

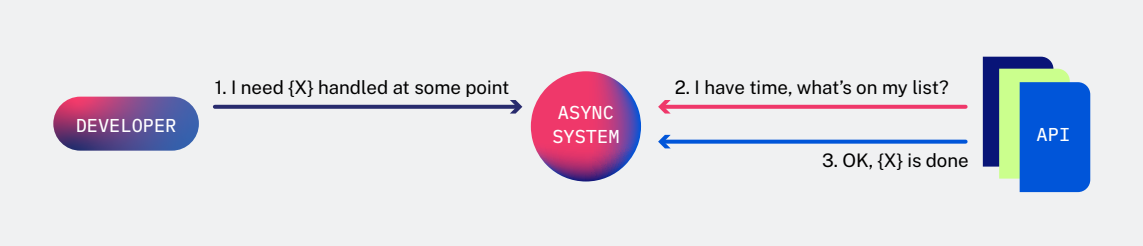
First, the developer sends a task to the asynchronous system.



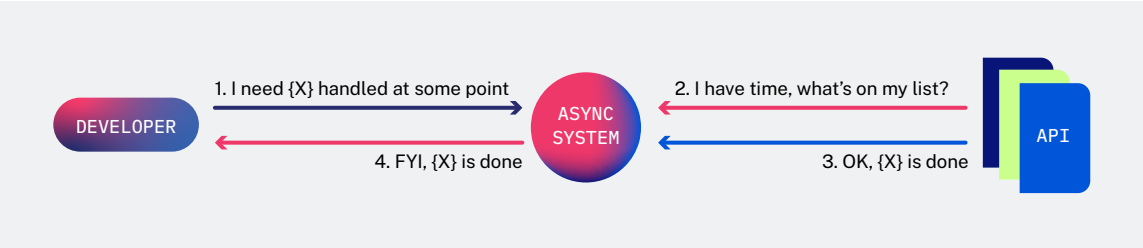
When the service is available, it can fetch the task from the asynchronous system.



The service notifies the asynchronous system that the task is complete.



Finally, the asynchronous system notifies the sender the task is complete.



For a less technical explanation of the difference between synchronous and asynchronous APIs, consider the difference between calling someone on the phone, or emailing someone to ask a question.

When you call someone, you ask the question and get an answer in a matter of moments. This is a synchronous conversation. In comparison, if you send an email you must wait for the response when the person has time to get back to you. This is an asynchronous conversation.

Best Practices for Using Asynchronous APIs

Now that we've introduced asynchronous APIs, let's take a look at some best practices to keep in mind if you choose to use this kind of API.

1. Know When to Use Asynchronous APIs (and When Not To)

Before you decide to use async APIs, it's important to consider the use case. The right type of API for the project is not always going to be an async API. You will likely use a variety of different API types to achieve the end result.

Here are a few reasons to consider asynchronous APIs:

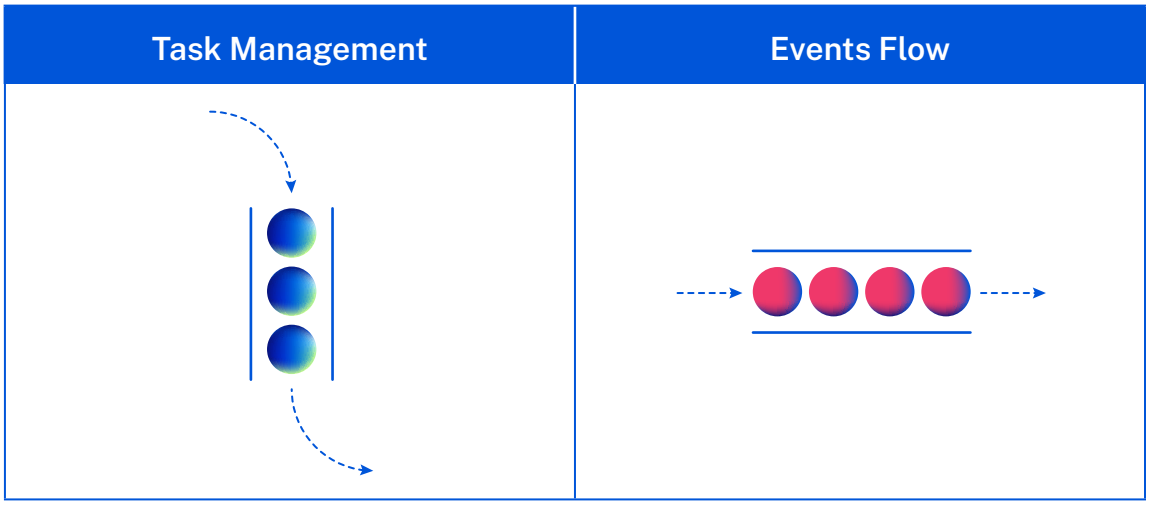
- Asynchronous processing of large tasks can avoid spiking the workload.
- Push model is made possible.
- New delivery and processing models.

And why asynchronous APIs might not be a good fit:

- Some workloads are inherently transactional and not a good match for the async model.
- Using asynchronous APIs can complicate onboarding for developers if they are not as familiar with the async model.
- Tools for async APIs might not be as developed (monitoring, security).
- Using asynchronous APIs requires a different mindset.

2. Choose the Right Type of Asynchronous API for the Task

Once you've determined asynchronous APIs are the right API type for the job, it's important to further refine your selection. There are different approaches to async API implementation, but for the purposes of this paper we will consider two of the most popular — queues and streams.



There are different considerations and technologies for each of these API types, even though they are all broadly categorized as asynchronous APIs. The table below breaks down the differences between queues and streams.

	Queue	Stream
Purpose	Storage service for messages between different services	Real-time processing of big data
Terminology	Messages, Consumers, Producers	Events, Publisher, Subscriber
Consumption Model	Read one item at a time on a FIFO basis, can re-queue	Chronological consumption, often in batches
Consumer Cardinality	Message consumed by a single consumer (load balance)	Events routed to all consumers (multiple effects)
Technology Examples	RabbitMQ, Amazon SQS	Amazon Kinesis, Kafka, Apache Spark

3. Define a Clear Message Schema, Structure, and SLAs

Developing your queue or stream shouldn't be done ad hoc. Like a REST API, it should be well documented and defined. If you don't define a clear schema, it will lead to further complications and challenges down the road.

AsyncAPI is a great example to reference. The AsyncAPI API specification is a method of defining the interfaces of asynchronous APIs. It is an attempt to standardize the format, much like OpenAPI does for REST APIs.

4. Build for Failure to Avoid Silent Failures

As you build an asynchronous API, it's important to plan for events or messages to fail as this will inevitably happen. If you don't build for failure, you risk a silent failure. This means the system is producing errors, but nobody knows anything is failing. You might only find out about the failures down the road when the magnitude is large.

For example, at RapidAPI we improved the billing process for our users by implementing a failure identification mechanism in asynchronous APIs.

There are two approaches that can help plan for failure:

1. Retry processing the failed objects.
2. Put the failed objects in a separate (failed) queue for human review.

You can combine these two processes, so the objects will be processed multiple times, and eventually moved to the dead letter queue if it is not successful.

Use Case Examples

Here are a few use cases that illustrate how you can use asynchronous APIs. Note again that you don't have to use the asynchronous model everywhere — in fact many transactions in your project will likely be synchronous. However, asynchronous APIs are a powerful addition in certain use cases.

Streams: Streams are a popular way to handle analytics. For example, you can send analytics events (logs) to a stream. The stream will be able to process these events and return data to an analytics processing system.

One benefit of using streams this way is the ability to handle large spikes in volume without bringing down your system or losing analytics.

Queues: Using a queue based approach is one way to handle large file uploads that you might not be able to process in a second or two.

In this scenario, the file will be pushed to a task queue. A file uploader then takes the file from the queue, and you can be notified when the upload is complete.

Using queues this way enables you to handle large file uploads without blocking the entire system.

Using Kafka-Based APIs

If you're ready to get started with asynchronous APIs, RapidAPI recently released the ability to publish Asynchronous Kafka APIs on both RapidAPI Enterprise Hub and the public RapidAPI Hub. This allows developers to manage both synchronous and asynchronous APIs from one centralized hub.

It is easy to get started and test Kafka-based APIs directly from your browser — you can see a demo Kafka API on the RapidAPI Hub **here**. You can also **add your own Kafka APIs** to the RapidAPI Hub or RapidAPI Enterprise Hub. Check out our blog for more information or contact our support team with any additional questions.

Apache Kafka: A Leading Enterprise Choice

Apache Kafka is a distributed streaming platform that allows building real-time streaming pipelines and applications. It was built by LinkedIn in the early 2010s.

Kafka is increasingly used as the message broker in event-driven architectures with asynchronous microservices. Kafka clients allow you to write distributed applications and microservices that read, write, and process event streams.

Popular use cases for Apache Kafka include messaging, event sourcing, and log aggregation.

How to Get Started with Asynchronous APIs

Kafka is increasingly used as the message broker in event-driven architectures with asynchronous microservices. Kafka clients allow you to write distributed applications and microservices that read, write, and process streams of events. Kafka has been adopted in more than 80% of Fortune 100 companies (source), as well as many of RapidAPI's enterprise customers.

Asynchronous APIs can be a powerful addition to your application. When determining whether or not asynchronous makes sense for your project, it is important to keep in mind these four best practices:

1. Know when to use asynchronous APIs (and when not to)
2. Choose the right type of asynchronous APIs for the task
3. Define clear message schema, structure, and SLAs
4. Build for failure to avoid silent failures