Best Practices for API Mocking

SMARTBEAR

Contents

Intro to API Mocking

What is a mock?

High quality. Fast. Cheap. Safe. In the ideal scenario, development, testing, and operations teams work harmoniously to deliver high-quality applications, on schedule, under budget, and error free.

But as applications become more complex, and companies adopt Agile methods (characterized by short cycles and iterative sprints), maintaining these ideals become increasingly difficult. Mocking provides opportunities for both developers and testers to build sandbox environments and "mock," or simulate, how services behave – this ensures high quality throughout the entire software development lifecycle.

Mocks, stubs, virtual services – which one do I use?

Mocking is one of those terms that cover a lot of ground, so let's be clear on what we mean.

The most common term for creating simulated components is mocking, but others are also used, and partly apply to different things: stubbing, mocking, and virtualization (or virtual services).

1. **A stub:** A minimal version of a service or API, usually configured to return predictable or even hardcoded data. The response is often coupled to the test case or suite, and the tests are going to depend on that data. It is used often when the suite of tests is simple. Stubs are often handwritten; some can be generated by tools for you.

2. **A mock:** Usually verifies outputs against some expectations, which can be set in the test. It is most useful when you have a large suite of tests and each of the tests needs different responses or data, or you want to test different scenarios. Maintaining a stub in that case could be manual and costly, so you can use a mock instead. Mocks often focus on interactions and are usually stateful; for example, you can verify how many times a given method was called.

3. **A virtual service:** A stand-in for a service and is created through virtualizing a service definition, e.g. OAS definition, or created by recording traffic from an internal or external endpoint.

For the latter, consider it a mock on steroids. It is a virtual service that works similarly to the real API you create. It imitates the real API, defines operations that clients of your real API will call, receives client requests, and simulates responses. It establishes a common ground for teams to communicate and facilitate artifact sharing.

These services are often shared with other development teams and frequently support more than one protocol (HTTP, MQ, TCP, etc.). They can be called remotely (over HTTP, TCP, etc.), whereas stubs and mocks most often work in-process directly with classes, methods, and functions.

How are these used in tests?

Developers are usually required to test that the service or application they are working on integrates with other system components via APIs. Frequently, they are unable access those systems during development due to security, performance, ownership, or maintenance issues that make them unavailable – or they simply might not exist yet.

We often see examples where a service or dependency is external and not suitable to run tests against (no sandbox environment or making requests results in costs not budgeted for).

This is where mocking comes in: Instead of developing code with actual external dependencies in place, a mock of those dependencies is created and used. You can make the mock "intelligent" enough to return similar results to the actual component it simulates, thus letting development move forward without being hindered by unavailability of services or systems.

As the software development lifecycle progresses, the need for more intelligent and dynamic responses requires a more intelligent mock. This is why we often see virtual services used for system testing where stubs and mocks for unit/module/acceptance testing.

Virtual services are often deployed in an environment where they can be shared and used by many teams simultaneously, whereas stubs would be individual instances per team.

Mocking vs. contract testing

Contract testing is a technique used for testing an integration point by checking each application in isolation. It ensures the messages it sends or receives conform to a shared understanding that is documented in a "contract."

For applications that communicate via HTTP, these "messages" would be the HTTP request and response, and for an application that used queues, this would be the message that goes in the queue.

In practice, a common way to implement contract tests (the way Pact does it) is simple. Check that all the calls to your test-double returns with the same results that would come from the real application.

But wait ...this sounds like mocking. True, they are both ways of simulating requests and responses, and testing how components of your services interact. But there are differences.

Let's consider how mocking and contract testing can be used at different points of your software development lifecycle.

When would I use contract testing?

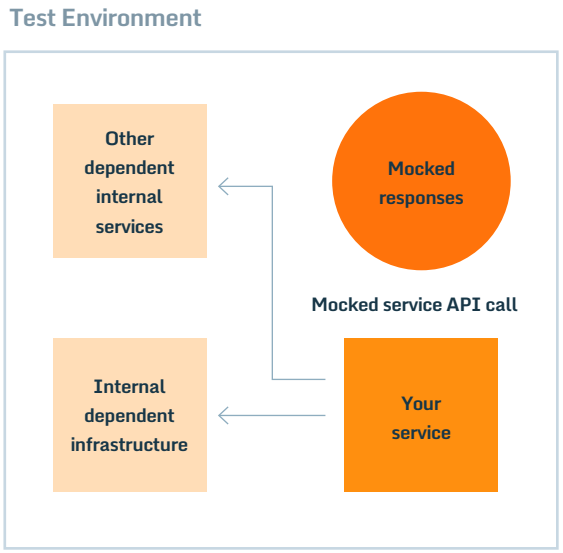
Contract testing is immediately applicable anywhere where you have two services that need to communicate – such as an API client and a web frontend.

Contract testing really shines in an environment with many services (as is common for a microservice architecture) are being developed in parallel and need to be tested at speed. Having well-formed contract tests makes it easy for developers to avoid version hell. Contract testing is the killer app for microservice development and deployment.

When to mock

There's an array of use cases for mocking. At a high level, mocking should be used to help conduct different types of testing, thus increasing quality and speed. Here at SmartBear, we recommend mocking when your service is not developed yet or where you use internal or external dependencies like system calls or accessing a database that is not readily available.

For the next chapter, we'll talk about how mocking and contract testing work as part of your development workflows, plus examine some use cases based on our experiences with customers.



API-first development & API mocking

How do mocks and contract testing accelerate API “design first”?

Contracts are fine as written definitions or documents, but an API is a business asset. And you and your teams (and partners) are going to want to demonstrate it and try it out.

Mocks and contract testing are important in validating the premise of the contract and ensuring that the requirements are implemented.

Both API-first and API design-first emphasize the importance of documents and contracts. They provide a reliable source of truth for API design and development teams. Both approaches have stakeholders getting involved early in the design process before writing any code, providing several advantages.

Let’s explore these

Teams can work in parallel, using the API definition to create and mock APIs, which allows producers and consumers to try out API designs and test APIs before deploying them. You don’t have to wait on other teams to complete different phases of the API.

You can use API definitions and mocks, especially when hosted in tools like SwaggerHub, to automatically generate things that improve the developer experience, like interactive documentation, mocks to try it out, client libraries, and SDKs.

Why should I care about mock “drift”?

In the Agile world of software development, the pace and rate of change of services are fast, and it’s a challenge to keep your mocks up to date, especially in a complex microservice architecture.

Where API definitions and mocks are not integrated and when your mocks and your live services are deviating from one another due to rapid development updates, we call this mock drift. This is a great use case where contract testing helps you keep pace with the changes in your service and reduces your need to keep mocks updated at a microservice layer.

You may then focus your efforts to update mocks as part of less frequent integration tests. This way, you leverage contract testing and mocks as part of your development flows and teams are able to accelerate with the right kind of test at the right time.

The mock is updated every time you save your API. The mock helps you test your API when designing it, that is, before it is implemented by developers. Also, the mock allows developers to start building client applications even before the API backend is ready.

Instances that help speed

Some examples of where mocking accelerates development and improves collaboration:

- **The design phase of an API.** You can quickly collaborate with stakeholders and apply changes to refine the API contract without waiting for a server deployment or the development team's availability. With mocking, you can try out the changes and ensure they work as expected.
- **Collaboration and reduced dependencies.** This is helpful between development teams, especially your frontend and backend. It's perfect when API is still under development, and you want to avoid bottlenecks for multiple teams.
- **Making your APIs available to others.** Before they commit to using them, that is. With mocking, you can provide a simulation of the actual API for testing purposes without the need to give access to your product or provision credentials. Demos can benefit from API mocking in the same way.

A simple case study

A development company has been developing microservices. They used OAS-based APIs to communicate between them. Teams used to have to wait for one another to finish the APIs before starting work. This dependency slowed down projects and caused stress.

Using API mocks, they could simulate their API schemas and defined behaviors. They developed both sides of their service in parallel, without having to wait on code to be written before testing can happen.

Real-life mocking scenarios

Developers are typically tasked with writing code that integrates with other system components via APIs. Unfortunately, it might not always be desirable or even possible to access those systems during development.

There could be security, performance, or maintenance issues that make them unavailable. We frequently see teams dependent on services owned by other groups, but they are not accessible for testing or modifying. They're either in production, or say, an external service needs to be called, but it's not available or suitable for testing.

This is where mocking via recording comes in: When functional testing a component that depends on external APIs, mocking can be very useful.

For example, you might need to test a geo-location functionality in your component that in turn uses Google Maps APIs to coordinate lookups. Instead of using the actual Google Maps API, you could use a corresponding mock that returns known results to your component.

This helps you validate the results from your component, as they are not affected by eventual inconsistencies in the external components. So an added benefit to mocking is that you keep your usage quota of external APIs to a minimum.

You may need to test against your API gateway as part of application testing – but the gateway APIs do not have a sandbox. We frequently see customers citing systems owned by production that are not made available to testing and cannot be modified to reflect different test cases.

This dependency (and lack of availability) causes delays for test teams and leads to frustration. Requests to internal and external services during test runs also can cause other issues:

- Tests failing due to connectivity.
- Test suites running slow due to networking and access.
- Hitting third-party API rate limits.
- No sandbox or staging server for test cases.

Mocking via recording means recording HTTP or HTTPS traffic to some existing API and creating a virtual service from the recorded requests and responses it detected. Another approach is to record a live interaction and replay it back during tests. This mock can then be configured to support the test use cases and run as often and wherever required to support testing.

Build out a test project – virtually

With SmartBear solutions, you can configure the route options property and set the functioning mode of a virtual service. You can configure your service to forward all incoming requests to a real API and create virtual operations for those requests.

Once you have the recording created and configured, you can start to build out the other parts. The recorded mock can be left to run as often and as long as you need, and your team can utilize it as part of unit or integrations tests.

If you want to validate your mock has not drifted, incorporate contract testing against it and monitor changes to the service from what your producer or consumer expects.

Then use your API mock in different scenarios.

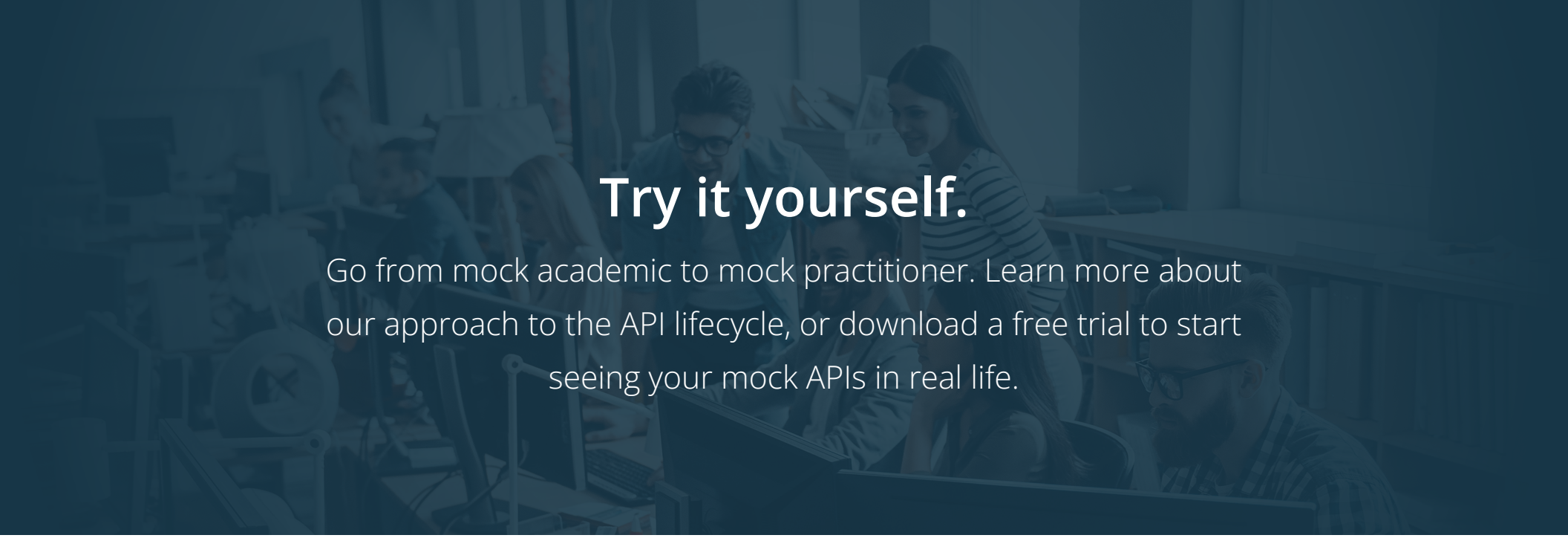
- External dependencies – Testing unexpected behavior and third-party dependencies.
- Isolated development – Development environment for use by operation and architects.
- Early functional testing – Starting testing work early in development, even before actual components are available.
- Customer evaluation – Providing evaluation and pre-sales opportunities.
- Avoiding API dependencies – Developing against future or unavailable functionality.

As you work through the software development process, you can use mocks at different stages to test different scenarios and update your mocks to reflect what you expect real-world conditions to be.

These capabilities provide real value to your teams and enable development and testing to progress in parallel. They also provide the following benefits:

- Sandbox environments to modify as you need
- Multiple versions to support different test use cases and outcomes
- Lower cost of running tests against rated external services
- One environment to establish a source of truth
- Monitor traffic to the mock and analyze behaviors to validate expectations

While these are more basic examples, we hope this book provides insights into using mocks in development. If there are any topics you want to discuss further with us, please contact us and speak with our Solution Engineers.



Try it yourself.

Go from mock academic to mock practitioner. Learn more about our approach to the API lifecycle, or download a free trial to start seeing your mock APIs in real life.

SMARTBEAR SwaggerHub

The Single Source of Truth for API Development

Create Free Account

SMARTBEAR ReadyAPI

The All-in-One Automated API Testing Platform

Start Free Trial

PACTFLOW A SMARTBEAR COMPANY

The Most Comprehensive Contract Testing Platform

Try for Free


SMARTBEAR