**LEARN API TESTING – OREILLY – BOOK**

1. A microservice is an API that deals with a single requirement.

An API abstracts the application layer and provides the resource(s) for consumption by the client. APIs are the backbone of any typical web application, multi-tier web application, or mobile application that hides the inside details of the system, such as how an online payment is processed for a consumer.

The API reads the data from the back end based on the user requirement/request and sends the response to the front end.

Figure [1-1](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fig1) shows a typical service-based software application architecture. It has a database at the back end, APIs in the middle tier, and requests made from a browser or mobile application. We will discuss this setup in detail in the next chapter.

Chart, diagram, box and whisker chart

Description automatically generated

A typical web application[**1**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fn1) can be an e-commerce application, where the user wants to see various product offerings and then buy a product as per their needs. Requests are typically made from the front end/GUI. The middle tier has various components in the form of APIs, such as an API for listing the products based on the requirements of a user, another API to add the product to the e-cart, and another set of APIs or third-party payment APIs to deal with the payment processing on behalf of the e-commerce web store.

A microservice is an API that deals with a single requirement and the service can be functional/deployed independently. Microservices[**2**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fn2) are APIs that define the business logic of a typical software application and fulfill the develop-fast-and-scalable software development philosophy. We will discuss this more in the next chapter.

API testing deals with business workflows. This may be categorized into black-box testing, but technically speaking, it is more of a gray-box testing where the tester knows some internal details of the implementation in brief, but not in depth. They test the APIs individually by having an understanding of the technical aspects of the code path or logic used inside the API.

*“Good to have internal knowledge of the implementation for a given API.”*

API testing is testing the end points[**4**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fn4) of the given API based on the given contract. The endpoint is defined in terms of the URI[**5**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fn5), such as /api/v1/products/{productId} or /api/v1/products. The contract should be in the required format (JSON/XML) of the request, and it may or may not include the parameter(s) based on the request method.

Accessing an API requires a mechanism that allows us to perform various actions based on the requirement(s), which are called request methods[**6**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fn6).

PI testing tests the middle tier before it is consumed by the consumer/front end. The tester makes sure that the endpoints are correct and they accept the request in the given format with required parameters and provide the correct response in the prescribed format. This testing directly deals with the application server. It may involve testing the individual component of the application or combining a few components to test a user workflow. All the standard testing techniques are performed while testing APIs, like equivalence class partitions, boundary value analysis, large requests, invalid requests, unauthorized requests, etc.

API testing requires specific tools, such as curl[**7**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fn7), Postman[**8**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fn8), and RestAssured[**9**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fn9), which support the request methods and the protocol that is used to retrieve the API. The commonly used protocol is HTTP(S)[**10**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_1_Chapter.xhtml#Fn10). The tester keys in the URL with the required request method and requests the parameters in the API testing tools in the same way as the consumer of the API and then verifies the response/output in the context of the application.

**A test plan is required, just like user workflow testing. The test plan has input, expected output, and a precondition.**

**Types of API Testing**

An API responds to a request by the consumer/front end.

The response should be quick.

The API should not be allowed to be accessed by an unauthorized user.

When concurrent users access the API, it should respond within the stipulated time.

Invalid requests to the API should be handled appropriately and an error message

should be returned.

The API should adhere to the local laws.

If the API is provided as a service, then it should maintain the contract with the consumer, the parameter should not change, and so on.

The following are the types of API testing:

* Functional testing addresses the functional aspects of the API, such as returning a response as per the business requirements.
* Performance testing addresses the response time under load. When multiple requests are made for the given API at the same point in time, the API should return the response in the allowed time limit as per the SLA definition agreed upon between the service provider and consumer.
* Security testing addresses the unauthorized access of the API by gaining access to the session, parameter tampering, and so on. The API should not allow any anonymous/unauthorized users to gain access to the data via itself.
* Noise testing addresses invalid or malfunction data in the request. The API should respond accordingly and on time. If the data is invalid, the API should respond with the proper error code/message.
* Error code and message testing address incorrect input data and responding with the appropriate error code and message.
* Scale testing is related to infrastructure, which is a DevOps routine job, but the API gets tested in this scenario as well. This is mostly the case in microservices architecture where a particular API is used more frequently. The API should be made scalable since the concurrent access shall be more frequent and the API should be made available all the time.
* Compliance testing falls in the local jurisdiction where the API is being consumed. For example, if the API is asking for personal information (cell number, city of birth, etc.), then this information should be protected by the vendor, any attempt to get this information should not be allowed, and audit logs should be maintained.
* CDCT (consumer-driven contract testing) means that the service provider always maintains the same request payload. This is critical for the business of the service provider. If the payload is changed, then the consumer request will start failing and it will be a loss to the business.

* 1. Easy to automate

* 1. Faster at finding bugs

* 1. GUI independent

* 1. Maximum code path coverage

### Designing Test Strategies

Testing a monolithic web application is easier than testing a microservice. To test a microservice, you need to implement additional stubs and/or mock the APIs for end-to-end testing.

A typical REST application architecture is shown in Figure [2-3](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_2_Chapter.xhtml#Fig3).

Graphical user interface, diagram

Description automatically generated

A REST service that meet the constraints of RESTful[**4**](https://learning.oreilly.com/library/view/learn-api-testing/9781484281420/html/525090_1_En_2_Chapter.xhtml#Fn4) architecture is called a RESTful service. The constraints are as follows:

* Client-server architecture
* Statelessness
* Cacheability
* Use of a layered system
* Uniform interface
* Support for code on demand

The client or the user interface can be developed independently without knowing the internal details of the server and its functions.

Statelessness helps in improving the overall performance of the server. The server is not required to know or maintain the state/session of the request. Its basic job is to provide the response without tracking the source with a session. This is achieved by the HTTP protocol.

Caching helps in improving the performance. If the same request is coming from various users, it can be cached. HTTP has a feature that helps in caching the responses. This helps the server to be more efficient.