In the field of software test engineering, API testing is one of the most important and sought after requirements for any application or system.

This document covers

- the aspects of API testing at operational level
- the challenges in API testing
- the features of an API testing tool
- design architecture of applications
- Api gateway

API testing

Functional (System)

Functional testing of an API is done to validate the correctness of the API response for a given request. For any given method, API response is what needs to be validated. Response is not only the response body, but also the returned status code.

Integration

Many times an API is meant to call another API or trigger another action. In these cases performing an overall call sequencing and validating them at the same time would be considered as integration testing.

Security

As the name suggests, security testing of APIs deals with the security of the API under test. Let it be about who is calling the API (client or other application), can the requested data be manipulated before reaching the server, and the response data is securely transferred to the requesting party. Validating the implemented security mechanisms like Basic Authorization, OAuth or 2 way authentication.

Performance

Performance testing of an API is done in various scenarios depending on the requirements.

- *Load testing - Calling an API for number of requests (Expected load) and monitor the responsiveness on various metrics like response time or throughput*
- *Stress testing - Calling an API for number of requests beyond the verified limit of the number of call an API can comfortably handle*
- *Endurance testing - Calling an API with a fixed number of requests for a longer period and monitoring the responsiveness*

In all these scenarios, it is vital to keep an eye while monitoring the responsiveness of the APIs for the part which is causing issues. End goal is always to figure out these issues, fix them, retest the same scenarios for the fixes and repeat until it is considered satisfactory.

API Protocols and Methods

Protocols

There are many different protocols available, but most widely used and accepted ones are:

REST

Representational State Transfer is a web architecture concept, which represents the current state of the requested resource. Client or requesting entity makes a logical request with the required data for a specific resource to the server, server then responds with the current state of the resource without keeping or storing any information of the request.

REST follows these rules strictly

- Client-Server architecture
- Statelessness
- Cacheability
- Code on demand
- Layered system
- Uniform interface

Descriptions about these constraints can be referred to here.

SOAP

Simple object access protocol is a messaging protocol for web services. It uses XML to handle the request and response.
SOAP is a neutral and extensible protocol available for any programming language and other protocols such as HTTP, SMTP etcetera.

Methods

Available methods for API testing are:

- GET
  Most common and simple method which is to retrieve data from the server
- POST
  This method is to create a new resource in the system
- PUT
  This method is to update an existing resource, it requires that entire resource data should be sent as request body irrespective of the fields being updated
- PATCH
  This method is also to update an existing resource with an exception that it does not require entire resource data in request body, only the fields which need to be updated
- DELETE
  This method is used to delete an existing resource. DELETE is idempotent in nature meaning the result is same even if the same call is being made multiple times

There are other few methods available but the ones mentioned above are the most used.

API Testing Challenges

Psychology

Biggest challenge in API testing is the psychology of the people who have not yet done it or just started to do it. Making a shift from testing UI based components to API has a psychological fear that it is too technical in nature. Maybe beauces it doesn’t give the comfort of an interacting GUI and things happen at the back end.

All it requires is the basic conceptual knowledge of the client-server architecture. Web services functioning and a very good understanding of the system under test.

Documentation

Most of the time documentation for the APIs under test is either unavailable or not exhaustive. It fails to provide the relevant and required information to the tester. This leads to a lesser confidence in the tester even if the test coverage is fairly good.

Untouched Server Side

Things are happening at the backend and the tester is relying only on the response data. What happened and why it happened is not clear or not seen at least. Imagine having server logs where the tester can see the action happening and build confidence just by knowing what happens when she makes a simple api call.

Too Many Tools To Choose From

Market is flooded with myriad API testing tools. It is not easy to find out the best suiting tool for your context. POSTMAN is one tool which has proven to handle almost everything. Better to go with it than experimenting with others.

API testing tool

Must have features

- Option to select the verbs for the APIUT
- Option to provide headers
- Option to provide query parameters
- Option to provide request body (if applicable)
- Different environment for the same set of APIs
- Variable declaration for easy maintenance
- Option to collect related APIs in one set
- History of the used APIs
- Option to write pre-request script
    - Example - Catch an object from an API and use the value in subsequent APIs as variables
- Authentication
    - Basic

    - Oauth
    - Digest
- Selection of request body as JSON, text or XML
- Support for RPC calls

Good to have features

- Sharable link for a predefined set of related APIs
    - Compatible with other test tools
- Import of a new API from raw code
- Access to the request body from a file
- Writes tests
    - Example - Write tests to assert some logic, these tests can be used to ensure no failures with the integration of other APIs. If there are changes which affects the written tests, tests should be updated accordingly
    - APIs are handling a particular task as it is
        - Example - APIs are handling exceptions for data related issue
- Auto generated reports (emailable) in case of test success/failure
- History of last 10 (whatever config is possible) tests and results
- Run related and integrated API under test in one go
- Permission based access for a project/organisation
    - Example - Admin will have all the API access, like deleting a resource or creating a new one requires additional grant access
- Option to have global search
    - An api can be searched via its name or path
- Option to fetch the newly done changes for the tool
- Support for RPC calls, sooner grpc is going to be an industry wide accepted thing

Understanding the design architecture

Till now we have discussed the APIs and their usage, while going through we have understood that these APIs are the entry point or exposed component of a web service with a specific purpose. And these web services either individually or in combination form an application that we use on a day to day basis.

Understanding the design architecture for these applications in terms of how it structures the web services and other application components helps you visualize the skeleton and open a vast number of opportunities for you to explore. It provides you with a better understanding of each cog in the complete machinery.

Consider this analogy, you went to a burger joint to get a burger. As a customer you want to order a burger. You place your order (think of your order as a request to an application) with an attendant (thing of attendant as the entry point API of the application) at the dedicated counter (think of this counter as the dedicated host of the application or DNS). You give your details (think of details as request data) about the burger you want to the attendant, attendant takes the details and pass it to the next attending component (it could be a central dashboard of all the requests or a person taking all the orders and triaging, think of this component as load balancer at high level).
As a customer you get to know that your order will be available in 5 minutes and then you wait for it with the provided token (think of it as a session token). But for the attendant the understanding is different, she knows the exact working of how the burger would be made and presented.

She knows…

- *Who is arranging the buns*
- *Who is preparing the required patty*
- *Who is preparing the side items - French fries and soft drink*
- *Who is responsible for arranging the items into a burger, putting it together with the side item and then finally presenting it to the counter where the customer has been waiting for it*

She also knows the usual time taken by each component and hence an approximate time is conveyed to the customer. Which is fallible as the components performance may vary depending on multiple factors.

Key points to take away from the above analogy would be that

- Customer has limited information about the burger, he just knows the kind of burger he would get, taste, size and look of it. He is limited to change the details of required burger and expect a predefined burger in return
- Attendant on the other hand, has much more information about the entire process. She understands the system in detail as in how every component interacts with each other. With the information attendant has, it provides her with opportunities to explore the system at a granular level.

Point here is that we would be better equipped to be an API tester if we are being an attendant rather than a customer.

In order to be a good attendant it is necessary to understand the different patterns for structuring the application components.

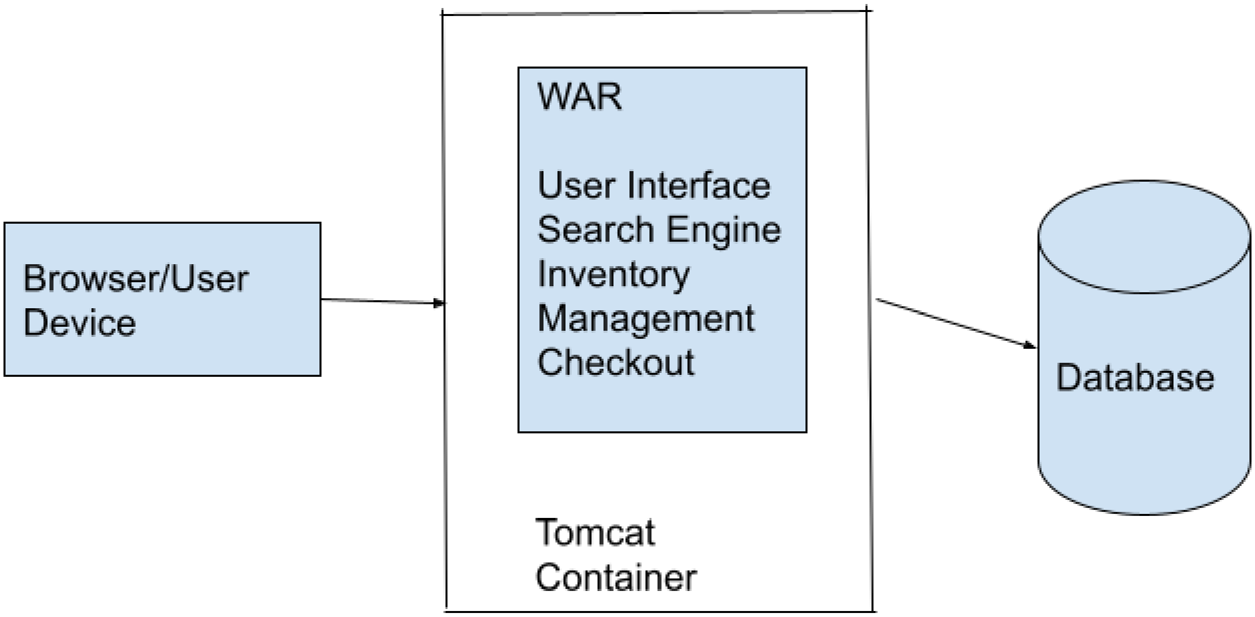
There are different structures in which these components are arranged, let's have a quick look at these structures/patterns

Before dwelling into the different patterns, for a reference point let's assume that we are developing an e-commerce application. For simplicity, we will assume the application has a User interface to search for an item, once searched option to add the item in the cart. In this, we would have components such as User Interface, search engine, inventory management and checkout.

Applications could be organized in different ways, let's have a look at them.

Monolithic Architecture

Application is deployed as a single monolithic application, where all the components of application (User Interface, backend services like Search engine, inventory management and checkout) are at one place, for example all the components deployed as one WAR file in a web container like tomcat.



As everything has pros and cons, monolithic architecture is no excuse.

Pros:

Monolithic architecture is very easy to:

- Develop - all the components being at one place it provides with less hassle to switch context and helps in faster development
- Test - For same reason it is faster to test the applications quickly
- Scale - Easier to scale, each running copy of application can be duplicated and deployed in parallel to scale and meet the demand

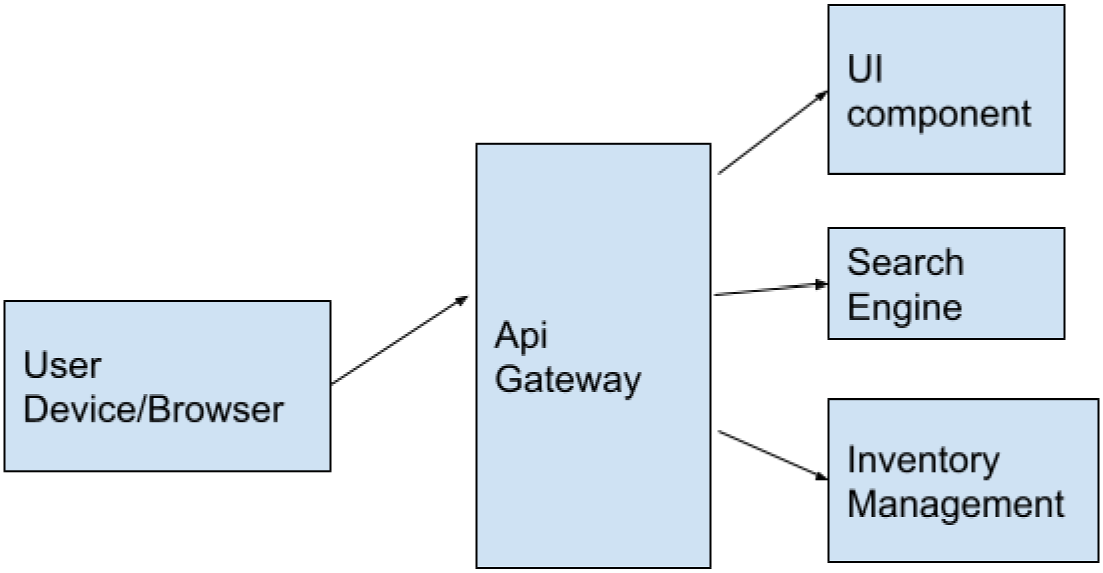
- Deployment - easier deployment, all components in one place. From the above example, just update the WAR file and rerun the tomcat. Deployment is done

Cons:
Once the application grow in size, existing pros tends to turn into cons

- Developer works on IDE to write the code, once the application grows, code base also grows and hence developer gets a slow IDE, impacting the performance of developers
    - For same reason, it becomes harder to debug and pinpoint the issues
    - Larger code base also intimidate the new developers working on the application
- Deployment becomes a task
    - From above example, larger application means larger WAR file, it will take more and more time for the application to start up and hence increases the deployment time
    - Entire application needs to be redeployed even if there is a slight change or update in one of the components
- Scale is not easy
    - It becomes very hectic to scale individual components. Individual components may require different resources, some might be CPU intensive some might be memory intensive. It becomes hard to identify this and scale accordingly
    - Teams working on components need to interact with each other and manage the changes and updates. This also reduces the overall performance. It is ideal to divide the teams component wise and application as well
    - Monolithic architecture requires long term commitment to the tech stack. It is not feasible to simply adapt to the newer technologies for a component. As it might break the other components or not be compatible at all
- One faulty component can bring down the entire application

Microservice Architecture

In micro services architecture all the components are deployed as micros services which in turn talks to each other for any logical requirements. From the above example, all the components such as User interface, search engine, inventory management and checkout would be deployed as separate services.



Pros:

- Independent deployment and changes
  - Micro services architecture allows developers to work independently, enhancing the performance and faster go to market cycle
  - Maintainability is easy
- Quick to understand
  - Smaller services are easier to understand
- No long term commitment to tech stack
- Debugging and fault isolation is much easier, allows developers to work freely to enhance the performance of the services

Cons:

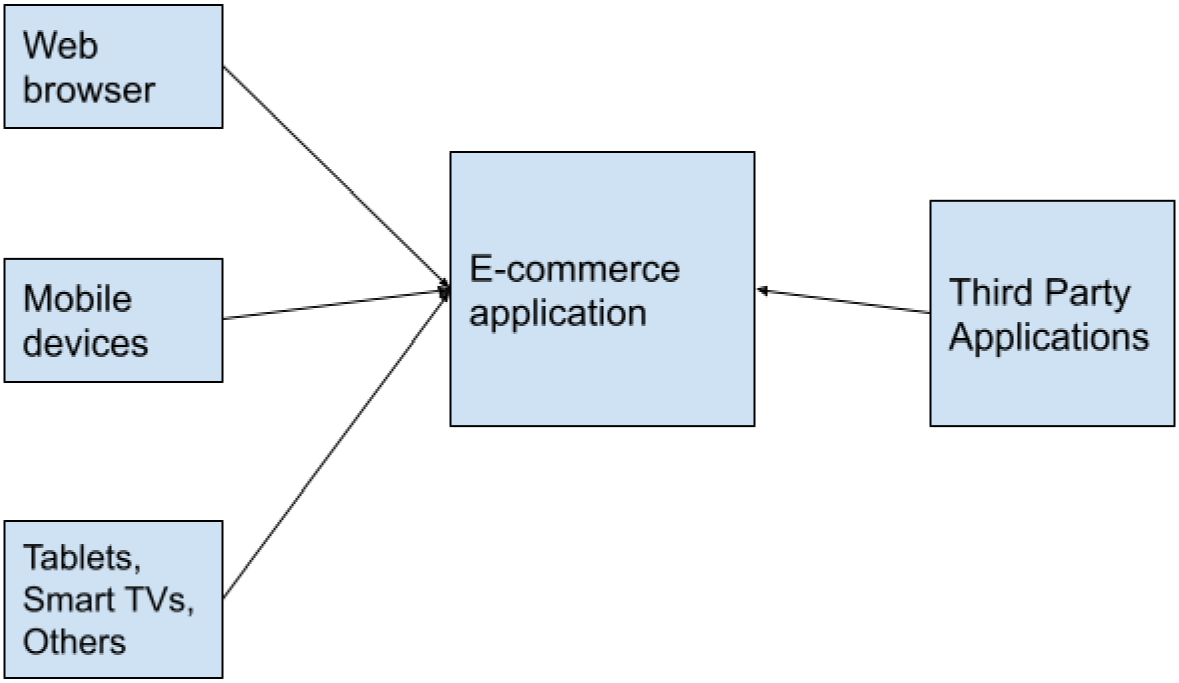
- Distributed system
  - Developers have an extra overhead of handling distributed systems. It adds lots of complexities as compared to monolithic architecture
    - Inter communication between services
    - Fallback mechanisms
  - Added operational costs of maintaining and deploying distributed systems
  - Increased memory and infrastructure

API Gateway

Before discussing the API gateway, let's understand the system where an API gateway is needed. During our discussion we have assumed an imaginary e-commerce application with different components. In real life we all know that in general an e-commerce application is available at different platforms, you can access the same application in your web browser, your mobile device and other devices like tablets or iPads.

As a user if you come to the application in your web browser, then you would be able to see lots of information for a searched item along with other related information while in your mobile application the information might be less.

Also, you might have noticed that the applications or sites you visited earlier are now showing up in your social media feeds as well. So what and how is it happening?



Above image summarizes the scenario we just discussed.

Discussion points are:

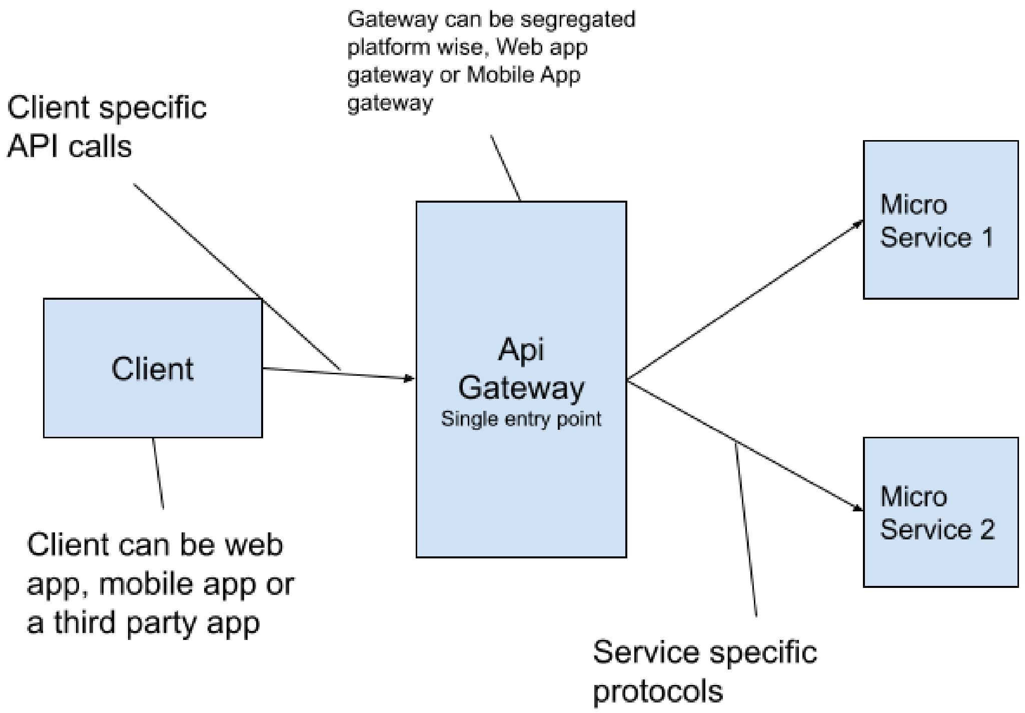
- Does the application has specific set of micro services/components for each platform
- Does application has APIs which fits all the requirements - One API, Fit ALL scenario

Having a One API and Fit all scenario is very complex and having a separate set of services for each platform is too costly.
In general applications have common micro services with ability to provide any permutation of data required in its scope. It is designed to provide a different set of data points based on the request made. For example a web application requires much more information for a product details than a third party app (like a social media - instagram). Third part app might only require product name, pricing and hyperlink to navigate to the application itself.

In short an application gets requests with varying inputs and it is required to provide the appropriate data. An API GATEWAY is a component which acts as an entry point to an application.

Api Gateway does computation and forwards the request to appropriate micro service. Sometimes it simply acts as a proxy and passes the request as it is.

As discussed micro services might use various communication protocols to interact. Hence, api gateway might also be required to take input from platforms in more known/common protocols and convert them into specific services protocol.


Gateway can be segregated platform wise, Web app gateway or Mobile App gateway
Client specific API calls
Client
Api Gateway
Single entry point
Micro Service 1
Micro Service 2
Client can be web app, mobile app or a third party app
Service specific protocols

Pros:

- Abstracts the application components from the client
- Discovery for services and components overhead is reduced
- Common and web friendly protocol for communication
  - Abstraction from the specific protocols used by the services internally
- Saves time for clients as they need to interact with one component which in turns take care of distributed call internally
  - Reduces the complexity at client by having computational logic in API gateway
- Specific API for clients for faster and optimised communication

Cons:

- It adds to the complexity of the system by having one more component to manage
    - New updates
    - Deployments

Kong gateways is the most popular open source API gateway.

Communication Style

In a microservice architecture, various services need to interact with each other. There are various communication styles available, which one to use depends on the context of interaction between the services. Let's have a quick look at those.

Remote Procedure Invocation:

Micro services need to interact with each other in real time and require tight coupling. Where any requests made have to be served in real time. Synchronous communication is important to the functioning of the application.

RPI could be used to solve the above problems, request/reply protocol is used by client and service.

RPI examples are:

- REST
- gRPC

Messaging

Messaging communication style is used when inter-communication services do not require synchronous communication. Communication is asynchronous and various sub-styles are:

- Request/Response - Response is required by the client once a message is sent
- Notification - No response is required, message is send
- Publish/Subscribe - Single message is published to one or more consumers
- Request/Asynchronous - Message is published and a response is required eventually

RabbitMQ and Kafka are the examples of most popular messaging queues

What does API do & various test validations

Hopefully we have gone through the documentation and understood the various components of a working application. There could be innumerable ways an API is being implemented, but based on what we have learned we could identify a few scenarios of how an API is being used and key points to keep in mind while doing API testing.

As a generic understanding, API calls are made with a request to get a suitable response, what an API does when a request is made is something that we need to understand. Understanding the actions when an API call is done helps identify granular details and unearth better and more test coverage

API call writes data to a database and displays the result

It involves an API which makes a call to a web service and the service uses data from the API and writes the required information in a database.

Various connecting points in the scenario are API, web service and database. To test such APIs testers should keep in mind all the connecting elements.
Various validations on such API

- *logs of web service,*
- *database queries and data validations*
- *Validations based on the API signature*

API call writes data to messaging service

It involves an API which makes a call and write a message to a specified messaging service

Various connecting points in the scenario are API, web service messaging service like (rabbitMQ & Kafka). The service which is going to consume the published message.
Various validations on such API

- *Message being published*
    - *Type and details of a message*
    - *Data validations of messages*
- *Validating the messages through messaging service itself*
- *Nature of messages*
    - *Duration*
    - *Single consumer/Multiple consumer*

- *Validation of logs - web service whose API is called and web service which consumed the message*
    - *Validation of any action which happens post consumption of the messages*

API call consumes data from a messaging service

It involves an API which consumes the messages from a messaging service, and uses the message content as the request input. There could be numerous possibilities about what the API does post reading the message. It is vital to understand the purpose of the API and decide the testing areas accordingly.

- *What does API do post reading the message*
- *Various data validations for varying messages*
    - *Wrong message*
    - *Incomplete message*
- *What happens if the API consumes same data again n again*
- *Logs of the service for which API is consuming the messages*

API call invokes another process from other service/API

It involves an API which takes some data as input, performs some action and makes a call to another API.
For example, search api takes a search query as input, displays the result for the query and in parallel makes a call to the inventory API to get the details of available inventory ready.

Such APIs are the most complex one as it comes with much more interacting elements. Like, API itself, web services or APIs which are being called, logs of API and other services being called, any Database operation
Various validations on such API

- *Various data validations for the API in hand*
- *Validations of actions taken by API*
    - *Direct action taken by API, like DB operation*
    - *Call to other web services or APIs*
    - *Logs of API being called and other subsequent services/APIs*

API call send the request to an API gateway

This call is directed towards an API gateway like KONG, which identifies the incoming request and computes the action that needs to be taken.

Various actions that could be taken:

- *Gateway acts as proxy and simply directs the API to a web service/API*

- *It works as load balancer and sends the call to various web service instances*
- *It computes the API call and decides which all service and API need to be called internally*
- *It might reject the API call if the call is invalid or gateway doesn't know what to do with the call*
- *It might give custom error message in case underlying web service is unavailable*
- *Tester needs to understand the purpose of the gateway completely. Above mentioned are just a few common scenarios.*

Would like to conclude by highlighting the importance of knowing the system in depth is what makes you a better backend/API tester.
Understanding the latest technologies and their usage would help you alot in understanding the end to end flows and decide on the run what all cases need to be covered for better coverage.