AT*SQA

API Testing: REST API, gRPC, and graphQL Micro-Credential

Syllabus

AT*SQA
MICRO-CREDENTIAL
API Testing
REST API, gRPC, and graphQL

Table of Contents

General Information

KEYWORDS

API Headers, Data Contracts, HTTP status codes, REST API, RESTful API, gRPC, graphQL

LEARNING OBJECTIVES FOR API TESTING: REST API, GRPC, AND GRAPHQL

Introduction to Quality Characteristics for Representational State Transfer (REST) API Testing

Understand what a REST API is

Functional Testing

Understand what a REST API data contract is
Understand what a REST API header is and what might be included in an API header call
Understand REST API response validation
Understand REST API data handling
Understand REST API versioning and what triggers and API version change
Understand the differences between paths and parameters
Understand factors in good API test design

Non-Functional Testing

Understand what should be covered in non-functional API testing

Introduction to gRPC and graphQL

gRPC

Explain the difference between gRPC and REST
Explain the difference in error codes between gRPC and REST

graphQL

Explain the purpose and use of graphQL

Quality Characteristics for Representational State Transfer (REST) API Testing

Introduction

REST APIs are one of the most commonly used in industry today. REST stands for Representational State Transfer. It is a set of constraints as opposed to a protocol or standard.

Returning to the 'What is an API' analogy of a game of catch, a REST API might only accept a tennis ball that is yellow – rejecting other balls such as an orange tennis ball or a black billiard ball. When the REST API receives the yellow tennis ball, it would process the ball (a message) and return a reply.

One characteristic of a REST API is that the messages (sent and received) are predominantly in the JSON (JavaScript object notation) format. If the reader is not familiar with JSON, it is suggested they become familiar with it.

A formal definition of REST API is as follows:

REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways.

When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information, or representation, is delivered in one of several formats via HTTP: JSON, HTML, XLT, plain text, and more. JSON is the most generally popular file format to use because, despite its name, it's language-agnostic, as well as readable by both humans and machines.

Headers and parameters are also important in the HTTP methods of a RESTful API HTTP request, as they contain important identifier information as to the request's metadata, authorization, uniform resource identifier (URI), caching, cookies, and more. There are request headers and response headers, each with their own HTTP connection information and status codes.

In order for an API to be considered RESTful, it has to conform to these criteria:

- A client-server architecture made up of clients, servers, and resources, with requests managed through HTTP.
- Stateless client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected.
- Cacheable data (i.e., data that is in fast-access memory) that streamlines client-server interactions.
- A uniform interface between components so that information is transferred in a standard form. This requires that:
    - Resources requested are identifiable and separate from the representations sent to the client.
    - Resources can be manipulated by the client via the representation they receive because the representation contains enough information to do so.
    - Self-descriptive messages returned to the client have enough information to describe how the client should process it.
    - Hypertext/hypermedia is available, meaning that after accessing a resource the client should be able to use hyperlinks to find all other currently available actions they can take.
- A layered system that organizes each type of server (those responsible for security, load-balancing, etc.) involved in the retrieval of requested information into hierarchies, invisible to the client.
- Code-on-demand (optional): the ability to send executable code from the server to the client when requested, extending client functionality.

Though the REST API has these criteria to conform to, it is still considered easier to use than a prescribed protocol like SOAP (Simple Object Access Protocol), which has specific requirements like XML messaging, and built-in security and transaction compliance that make it slower and heavier.

In contrast, REST is a set of guidelines that can be implemented as needed, making REST APIs faster and more lightweight, with increased scalability—perfect for Internet of Things (IoT) and mobile app development. (RedHat, 2020)

Functional Testing

INTRODUCTION

Functional testing is designed to assess the ability of the application to provide the proper functionality to the user. It tests what the software does. A tester will need to keep a few things in mind when testing a REST API.

1. API Data Contracts
2. API Headers
3. Response Validation
4. Data Handling
5. API Versioning
6. Paths and Parameters
7. Test Design

Each of these is discussed in the sections below.

API DATA CONTRACTS

A contract is an agreement between at least two parties where there is benefit to each party. In API data contracts, the 'agreement' is the type of data the API service provider will accept as a request and the type of data the provider will send in a response. If there is a significant change in the data the API will accept or the data in the response, the 'contract violation' is considered a breaking change.

As an example, when something is purchased, there is agreement to use a certain type of currency for an agreed product. If the provider requires dollars as opposed to crypto coins, a 'request' to purchase a product will fail (a failure response) if the requestor attempts to buy the product in crypto. Similarly, an API might expect an 'age' parameter to be an integer (e.g., 30) instead of a string (e.g., "thirty")".

A more formal API Data Contract definition is as follows:

Data contracts are a set of definitions and constraints for the structure of the data that Intelligent Recommendations consume. To allow Intelligent Recommendations to ingest the data shared with it and provide recommendations, you need to adhere to the data contracts as described in this article.

(Microsoft, n.d.)

A data contract may contain multiple parameters, nested or un-nested objects, and different data types.

The data types can be as follows:

- String
- Number
- Boolean
- Array
- Null
- Object

(W3 Schools, n.d.)

Testing the request body data contract needs to be inclusive of the following:

- String parameter tests will need to validate different string lengths including an empty string and the boundary of 4M. (MIcrosoft, n.d.)
- Dates are often written as a string; hence, tests will need to verify the date time zone, the date format (i.e., yyyy-MM-dd:hh:mm:ss), and the date validity (i.e., 29 February is only accepted on leap year)

- Numeric parameter tests will need to include integers and decimal values (floats) with both positive and negative (less than zero) values. In addition, the tests will need to validate decimal values with different degrees of precision (number of digits to the right of the decimal point). Boundary values must also be considered.
- Boolean parameter tests will need to include the case of the parameter (True vs. true).
- Null tests will need to be included in other type parameter tests including strings, numbers, Booleans, arrays, and objects. Null may appear as null, "null", or as an absent parameter. The tester will need to validate how each scenario is handled.
- Arrays parameter tests will need to include an empty array, an array with one value, and an array with multiple values.
- Object tests will need to follow the same criteria as a parameter tests – treating each object as a child or partial contract.

Data contract negative tests will need to include the following:

- Data type validation (i.e., using an integer such as 0 or 1 for a Boolean value).
- Invalid parameter key casing (i.e., {"Key": "value"} vs. {"key": "value"}).
- Missing parameters (i.e., the data contract calls for {"A":"a", "B":"b"} and the data contains only {"A":"a"}).
- Extraneous parameters (i.e., the data contract calls for {"A":"a", "B":"b"} and the data contains an extra parameter {"A":"a", "B":"b"}, {"C": "c"}).

A data contract can be considered "strict" where a missing parameter is considered null, and an extraneous parameter result in a failure. Documentation will need to be clear on the strict-ness of the data contract and the tests will need to be inclusive of strict-ness validation. A data contract can have strict-ness applied to the full data contract or parameters in the data contract. Each parameter will need to have "optional" or "required" flagged (in the data contract and API documentation) so the end user and tester can use and test the endpoint accordingly.

API REQUEST HEADERS

An API header might be thought of as icing (a header) on a cake (a request body). The icing 'surrounds' the cake yet is an important part of the cake. Icing is not required on a cake, but most cakes have icing. If a piece of chocolate cake is requested, the requestor might accept a piece of cake with chocolate icing but not a piece of chocolate cake with chocolate icing and nut sprinkles (especially if there's a nut allergy!). This shows how the header plays a vital role in the security and validation of the API call. In addition, the header provides information to the service receiving the request (the service providing the API). A header can provide information in both the request and the response. Conversely, a header can be different between a request (header) and response (header).

An API request and response will have HTTP headers that will need to be evaluated and tested.

A formal definition of an API header is as follows:

In API requests, request headers are used to provide additional information for a server to process an API request. For example, they might specify that the data being sent is in JSON format, identify which version of the API to call, or provide an API key for authentication. Typically, headers are used to provide some metadata related to the request, and don't directly specify which data you want to retrieve (that's handled by the API request URL). (Ana, 2023)

The most common parts of a header the tester will need to evaluate are the Authorization attribute, the Accept attribute, and the Content-Type attribute.

Bearer/JSON Web Token (JWT) Authorization HTTP Header

Where a web site might have a security layer requiring a user ID and password to get access, an API might have an authorization 'key' to allow (or disallow) API calls. A common approach to the authentication 'key' is a Bearer token or JWT. This JWT header token is part of the API header.

Authorization tokens tend to have an expiry date, so the person or process may need to periodically get a new token. In other words, if the requestor has a key, it can be used for a set period of time (e.g., 30 minutes) and the API calls will no longer accept a request after the token expires; hence, the user will need to get a new key after the expiration period has passed.

When testing a REST API that uses a JWT token for authorization, the tester will need to evaluate the token to validate that the correct token content is present for valid access and yet does not have too much access – creating a security attack pathway. An invalid token, or token that is missing needed access information, will result in a 401 (unauthorized) response from the REST API request.

A formal definition of a JWT token is as follows:

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it. (jwt.io, n.d.)

A tester will need to validate the content of the JWT token. Examples of content that needs to be validated might include the following:

1. An expiry date of the token. REST API calls made after the token is expired are expected to result in a 401 (unauthorized) response.
2. Access roles (i.e. database.read or database.write) where an absence of the correct role type will result in an HTTP 401 (invalid) response.

Accept Header

An accept header tells the API the data format of the response. It may be required to stipulate if the client can accept data in the form of XML, JSON, etc. Some APIs require the acceptance header and others use a default (often JSON). Some APIs support multiple data types for the response. If the API does not support the accept data type, a failure response is expected.

Here is a more formal description of an accept header:

The Accept request HTTP header indicates which content types, expressed as MIME types, the client can understand. The server uses content negotiation to select one of the proposals and informs the client of the choice with the Content-Type response header. Browsers set required values for this header based on the context of the request. For example, a browser uses different values in a request when fetching a CSS stylesheet, image, video, or a script. (contributors, 2022)

Testing should verify that a valid accept parameter is accepted and the expected output type is in the response.

Content-Type Header

A content-type header tells the API the data format of the request. It may be necessary to stipulate the request format, e.g., XML or JSON. If the server cannot accept the request data format, a failure response is expected. Some APIs require the content-type header and others use a default (often JSON). If the request does not have a payload (data), the API may not require a content-type.

A formal description is as follows:

The Content-Type representation header is used to indicate the original media type of the resource (prior to any content encoding applied for sending). In responses, a Content-Type header provides the client with the actual content type of the returned content. This header's value may be ignored. (MDN contributors, 2022)

Testing should verify that a valid content-type header parameter is accepted and is in the request.

RESPONSE VALIDATION

Any request is expected to be followed by an HTTP response, even if it is a '404 file not found'. The response may include the following information:

- HTTP status code
- Header information in the HTTP response
- HTTP message or message body

An HTTP request is always expected to have a response. (RFC, n.d.)

The tester should have an understanding and expectation if the HTTP response is asynchronous or synchronous. (mozilla.org, n.d.)

Testers should understand the common HTTP status codes and the HTTP status code expected for each test scenario. (Mozilla, n.d.)

In summary, the response should have an HTTP status code but may or may not have an HTTP header and may or may not have a response message. In fact, some providers will purposely not send a response message with an invalid request as a form of security.

As a tester, it is beneficial to either learn common HTTP status codes and/or have the HTTP status code reference close at hand. The Mozilla web site has a good reference of the HTTP status codes (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status). This document will give an overview of the common successful and error responses.

Positive (happy path) Validation (20x)

It is important for the tester to know if the response is expected to be synchronous or asynchronous. If the response is synchronous, the response is expected to include the processed data – completing the API 'transaction'. An endpoint with an asynchronous response will receive the request and send the response while processing is still underway. Endpoints that have an asynchronous response will often have a different endpoint to allow the user to query the progress or status of the work that was initiated with the first endpoint's call. In other words, with an asynchronous endpoint, the first call starts the 'transaction', and the second call gets the 'transaction' results – completing the 'transaction'.

For example, when buying a pre-made cupcake from a shop, it is expected that the cupcake is available now (synchronous). When buying a pizza from a pizza shop, it is expected that the pizza be ready at some time in the future (asynchronous). Either way, the product is requested and paid for, but the transaction is completed depending on the 'processing' required.

An example of an asynchronous API workflow might be as follows:

An end user wants to order a cake that is not yet made. The end user wants to call the API to order a new cake.

The endpoint is \cake\create with the following request body:

```
{
"type": "chocolate",
"icing": "vanilla",
"layers": 3
}
```

The request for cake might be a 202 Accepted with the following response:

```
{
"cakeInvoiceNumber": 1001
}
```

The end user would then query the status of the cake job with the following endpoint \cake\status with the following request body:

```
{
"cakeInvoiceNumber": 1001
}
```

And the response may then be a 200 OK with the following response:

```
{
“cakeInvoiceNumber”: 1001,
“status”: “mixing ingredients.”
}
```

An example of a synchronous workflow might be as follows:

An end user wants to order a cake that is already made and ready to sell. The end user wants to call the API to buy a cake.

An endpoint looks something like \cake\buy with the following request body:

```
{
“type”: “chocolate”,
“icing”: “vanilla”
}
```

The response from the buy request for cake would be 200 OK:

```
{
“result”: “You bought a chocolate cake with vanilla icing!”
}
```

The synchronous end user transaction is now complete with no need to call the \cake\status endpoint.

The above examples show different HTTP success responses in context of asynchronous vs. synchronous endpoints. In this context, the tester will not only need to validate the HTTP response but will also need to validate that the HTTP status makes logical sense.

Error Response Validation (4xx or 5xx)

Error response validation can be just as important, if not more important, as success responses. Error responses might trigger a failure depending on the system under test and the circumstance under which the error is encountered.

The following examples further explain this concept:

- System A is the system under test. System A has a REST API endpoint that may call System B as part of its business logic.
- System A has an expectation that no valid HTTP request results in an error. System A is expected to scale with increasing traffic and an error encountered would have negative financial or business impact.
- System B has the expectation that if the system is under load, the user may get a 429 (too many requests) response. The user is expected to retry the request later. There is no financial or business impact to the API provider or the business(es) consuming the API.
- The tester needs to validate that an error encountered with System B, is handled in System A in an expected manner.

From these two examples, error testing could include:

1. Error message validation (i.e., the response message for a 429 might include the text "system under load, try again in 5 minutes"). The error message would need to be robust enough for the end user to understand the error while not exposing any intellectual property or exploitable security vulnerabilities. An example of an error message that is unhelpful might be a response body that only consists of {"result":"error"}. An example of an error message with an exploitable response might be as follows {"result":"SQL server database SQL001.business.com is under load, notify bob@business.com of the issue"}.
2. Like a successful response, the HTTP status code will need to be validated that it makes sense under the circumstances in which the error is encountered. For example, the user would not expect a 400 "Bad request", when an unauthorized user is making the call (logically a 401 "unauthorized" would be expected).

3. Verification of the performance of the endpoint under defined business expectations. See the performance portion of this document for further information.
4. Documentation for the end users (or developers) so there is clear guidance when an error is encountered. For example, if there is a knowledge-based article for the API published, the knowledge-based article might include the time a user should wait before doing another call should an HTTP 429 be encountered.
5. Logging validation (i.e., the error is cited in a log store with clear analytics such as a date time stamp, a traceable error message, etc.).

Date Handling

Date handling is an important consideration for API testing. JSON does not support the date type. Therefore, dates often are either sent in the request (or response) as a string or a double (OADate). For example, a date might show as

```
{
“date”: “2021-12-30”
}
```

Hence, the string will need to be in the expected format including local(ization) and time zone. If an invalid format is used, an error can result (often from parsing the date). In addition, testing considerations will need to include daylight savings and leap year. The tester will need to validate the date time format and validate that the expected date time format is documented in the requirements and any support documentation. If API calls are logged, consistency between the date format of the API call (request or result) and the logging timestamp will help prevent confusion (i.e., eliminating the need to determine what the time difference is when assessing the timestamp in the API call versus the logging timestamp). Developers often use UTC to avoid time zones and leap year issues. If UTC is used as a datetime format, the logging should also use UTC.

Null Handling

Null handling is an important consideration for API testing. Null (or nothing) can be handled in several ways, so it is important for the tester to understand and validate null handling.

For example, a null value might be handled as an empty string,

```
{
“null-value”: “”
}
```

or a null value can use the value of “null”;

```
{
“null-value”: null
}
```

or a null value can be absent from the message;

```
{
}
```

hence, how a null value will be handled must be documented and validated.

Numeric Handling

A numeric value can be positive or negative, an integer, or a decimal (possibly a float); hence, numeric handling is also an important consideration for API testing. If a value has a precision that is not expected (i.e., 0.1 versus 0.10), the parsing of that value can result in failures. Therefore, it is important for the tester to understand and validate how different numeric values are handled (i.e., rounding or truncating).

An example is as follows:

```
{
“numeric-value”: -0.15
}
```

Might be evaluated to 0 for integers only, -0.1 if the precision is truncated, -0.20 if the value is rounded up, or 0.15 if the negativity is discarded.

Documentation of data handling is important for development reference, support reference, and quality (testing) reference. It is suggested that there is ‘one source of truth’ where data handling requirements are kept to prevent confusion and quality issues (i.e., a swagger API specification).

HTTP Verb (Method) Handling

Depending on the API, only a subset of HTTP verbs or methods will be supported by the endpoint. For example, an endpoint might support a GET method but not a PUT or DELETE method. Validation of unsupported method calls must be included in the testing. Documentation of supported methods will need to be clear as well as the handling of unsupported methods.

API VERSIONING

There may be multiple API versions supported at the same time. For example, an API path with two different versions might look like v1/users/{id} and /v2/users/{id} . The approach might also use a value in a header to declare the API version being used. A minor change such as an added optional parameter in the API may not require a version change. However, a major change or ‘breaking change’, such as a data contract change or required parameter, is considered a reason to change the API version.

For example, if there is an API /v1/users?id=1 with the required integer parameter 1 and there is a change for an optional parameter /v1/users?id=1&name=jo a version change may not be needed. However, if there is a required parameter or a data contract change to the ID parameter from an integer (/v1/users?id=1) to a string (/v1/users?id=emp001), a major API version change is in order.

The tester will need to keep API versioning in mind when designing, developing, executing, and reporting on API tests. For example, an optional parameter in v1 that becomes a required parameter in v2 would require tests that would fail if the required parameter is absent for v2 but pass for v1.

The tester will need to keep in mind the fact that APIs tend to evolve over time; hence, any configurability that can be added initially in the API test code will serve to help future proofing. In other words, when possible, try to think of how the API might change and code accordingly to reduce the need to rewrite test code for each API version.

When reporting on the API test results, the reporting will need to clearly identify the API version under test to prevent confusion in the reports. When testing multiple API versions at the same time, each API version test result will need to be distinguishable and auditable.

PATHS AND PARAMETERS

Depending on the API’s design, the URI may have different paths that may include parameters.

For example, an API can have a path such as \jobs\fulltime or \jobs\parttime. Further, an API might have an argument with a parameter such as \jobs\fulltime?industry=softwaretest.

When testing an API, both valid and invalid paths and parameters will need to be considered.

For example, a positive test for \jobs\fulltime might have the negative test for a path of **bogus**jobs\fulltime or \jobs**bogus**time.

A positive test case for a parameter \jobs\fulltime?industry=softwaretest might have a negative test of \jobs\fulltime?industry=**bogus**industry or \jobs\fulltime?**bogus**industry=softwaretest .

Arguments may include characters that are invalid in a URL. These characters might be valid if they are encoded correctly.

For example, an API call to \jobs\fulltime?industry=software test (space between "software" and "test") may be invalid but \jobs\fulltime?industry=software**%20**test might be valid.

The tester will need to develop and validate different positive and negative path and parameter scenarios. Error handling will need to be validated (see error response validation section above).

TEST DESIGN

When designing the tests for an API, the following should be considered:

- Structure of the API endpoints
- HTTP methods (GET, POST, etc.) used by the endpoints
- Validation of data both sent and received
- Error conditions and the resulting error messages
- Security
- The API consumer(s)
- Performance traffic expectation of each endpoint
- Asynchronous or synchronous nature of the API architecture

The functionality of the API endpoint(s) can be determined from the requirements, use cases, specifications or even conducting exploratory testing to learn about the application. Design tests for API endpoints considering both the functionality and use of the endpoints.

Non-Functional Testing

Non-functional testing concentrates on how functionality is delivered to the user. For a more complete discussion on non-functional testing, see [ISTQB_FL_SYL]. This section concentrates on the non-functional quality characteristics that are of primary importance in testing API applications.

PERFORMANCE TESTING

Performance testing for an API may have a high impact resulting in a higher risk and therefore higher testing priority.

An API can be one in a chain of API endpoints in a process. Hence, there is value in finding process bottlenecks (finding the weakest link in a chain). Slow performance can lead to error conditions that may or may not have been accounted for in the process consuming the API. It is important for a tester to evangelize performance metrics be included in requirements. Once the performance requirements are established, the tester needs visibility to performance issues as early as possible due to the 'deep' nature of many performance issues. In other words, fixing performance issues may require significant and expensive changes resulting in a conflict between fixing a performance issue and completing a deadline.

Performance testing can take significant time to complete foundation tasks such as test data creation, performance environment creation, performance node (the machines creating the traffic) creation, etc. A tester needs to keep in mind the foundation tasks when planning and scheduling performance tests as these may require significant time, money, and engagement with other people such as DBAs, network engineers, etc.

Changes to an already released API may trigger the need to execute another performance test pass. The tester will need to be in communication with the developers to understand if code changes are likely to result in performance changes.

Though performance testing is considered a specialty, it is suggested that a tester understands basic performance objectives and principles. A non-performance-specialized tester may need to design, develop, and execute “high-level” performance tests. If the tester needs to conduct performance testing, it is suggested the tester refers to the ISTQB Certified Tester – Performance Testing (CT-PT) syllabus.

API LOGGING TESTING

Logging plays a critical role in analytics, and defect triage. A robust logging architecture, can be utilized to train machine learning algorithms that can be used to forecast API traffic, identify traffic anomalies, find security exploit attempts, and more. (Levy, n.d.) Depending on the project needs, successful API call logging can be considered as important as failure logging.

Failure logging is intuitively important to identify and capture defects. Failure logging is also important for capturing timing issues - failures that only appear under certain circumstance or failures that happen at seemingly random intervals. API failure log analysis is important for capturing quality issues in both pre-production environments as well as production environments. When the tester does not have access to production logs, the system administrators often have the responsibility of giving visibility of log failures (via alerts or another process) and communicating the failures to the product team.

API logging may have different levels depending on the circumstances. For example, verbose logging may be used if an issue is being investigated whereas verbosity settings may be low during normal operations to save time and space.

An API log should have at least the following information:

1. The date time stamp of the logged event
2. Information such as a user ID to differentiate one user or process from another process that may be running at the same time
3. The HTTP status code of the event
4. A message to give enough information to a human (tester, developer, or data analyst) to get insight from the log entry for further debugging

The amount of log information needed depends on the project and data engineering requirements. (Tamura, 2015)

A tester's understanding of API logging allows the tester to both add quality insights early in the product life cycle (i.e., how are logs to be assessed for issues) and provides a foundation for analyzing issues during testing and after a product has gone to production.

API SECURITY TESTING

Security plays a critical role in the development and testing of APIs. Security can include security token validation, security token expiry validation, role-based endpoint access and much more. There are multiple tools that are available for use in security testing. (Tesauro, n.d.) However, security tools may require technical knowledge that equates to a high learning curve. As a result, many teams will contract a specialist security tester to perform security testing.

Before a security tester is approached, the team will need to decide on the following:

- The environment to be used for the security testing. The environment will need to be a "close to production" reflection of a production environment to provide the most value.
- The schedule of the security testing. The security testing will need to be scheduled with the API(s) in a near production state of development so the testing can identify any issues with close-to-production code. The security tests will also need to allow time for development to fix any issues that arise from the testing without compromising the intended release date.
- The frequency of the security testing. Depending on the release cadence and changes to the API code, or changes in the configuration in which the API executes, the security testing will need to be repeated.
- The freshness of a security test provider will need to be assessed. A change of security testing provider may provide a different security prospective or may uncover different attack vulnerabilities.
- Approvals for the security testing. Most environments require specific written authorization before security testing can commence to avoid a response to a simulated attack and to inform all relevant parties when the attacks will be simulated.

gRPC and graphQL

Introduction

gRPC and graphQL both have similarities to REST API where much of the same test approach (performance, successful versus failure, logging, etc.) can apply. Some differences in the API technologies are cited below.

gRPC

A formal definition of gRPC is as follows:

gRPC is a modern open-source high performance Remote Procedure Call (RPC) framework created by Google that can run in any environment. gRPC can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in the last mile of distributed computing to connect devices, mobile applications, and browsers to backend services. (About gRPC, n.d.)

A big difference between REST and gRPC is the use of HTTP/2. This dramatically increases network efficiency and enables real-time communication by providing a framework for long-lived connections. (gRPC on HTTP/2 Engineering a Robust, High-performance Protocol, n.d.)

An additional difference between REST and gRPC is the use of a .proto file that can be shared between the API service provider and the API consumer. This .proto file defines the data contract to be used by the data consumer. A change to the .proto file is a breaking change that would trigger the need to share the new .proto file with the API consumers. In other words, gRPC incorporates a .proto file to reference for the data contract – REST API does not have the advantage of a common reference file.

When you connect to a gRPC service, you will use grpc:// instead of http:// or https://.

The gRPC error codes are defined by a gRPC code – not a http status code as with REST. (Status codes and their use in gRPC, n.d.). All gRPC codes start with “GRPC_STATUS” and then contain plain English terms for the issue such as “Cancelled”.

gRPC use has an advantage with lightweight services (like microservices) where the requirements need low latency and high throughput communication.

A downside to using gRPC is that gRPC calls are not supported in a browser currently. Until recently (early 2023), most common REST API tools (such as Postman) did not support gRPC; however, gRPC tool support, including testing, is becoming more mainstream.

(James Newton-King, 2022)

GraphQL

GraphQL is a query language for APIs and a runtime for fulfilling those queries with existing data. GraphQL provides a complete and understandable description of the data in an API. GraphQL gives the consumer of the API the ability to extract what is needed and nothing more – making it easier to evolve APIs over time and enables powerful developer tools. (A query language for your API, n.d.)

GraphQL uses a query language ("QL") to retrieve specific data columns from the data source somewhat similar to tSQL but using a JSON-like language instead of a select statement.

For example, if there is a database table named "Users" with columns name "firstname", "lastname", and "address" and the tester want to query the "lastname" only in the table (in tSQL):

```
SELECT lastname FROM Projects
```

For GraphQL, the following message would be used:

```
{
  Users {
    lastname
  }
}
```

Take note that the parameters in a GraphQL query differ from JSON as each parameter is not surrounded by double quotes. This is an important distinction to keep in mind when learning about GraphQL.

GraphQL uses HTTP; hence, the error codes will align with HTTP status codes.

An advantage to GraphQL is its strongly-type schema. GraphQL has a Schema Definition Language (SDL) which helps determine the data that is available and the form it exists in. (Hitesh Baldaniya , 2021)

A downside to GraphQL is its need to perform a query (DSL) to fetch the data. Further, the GraphQL needs to be carefully designed so a single endpoint is created (where a REST API might have multiple endpoints with different responsibilities). (Esteban Herrera, 2022)

References

Acknowledgements

This document was produced by a core team from the AT*SQA Foundation Level Working Group – API Micro-Credentials

Judy McKay (chair).
Johnathan Seal – Author
Earl Burba, Judy McKay, Randy Rice - Reviewers

The core team thanks the review team for their suggestions and input.

AI (such as ChatGPT) was not used to create content for this document.

Purpose of this Document

This syllabus forms the basis for the Association for Software Testing and Quality Assurance (AT*SQA) Micro-Credential for API Testing. This particular syllabus is focused on the Introduction and the Test Planning and Design areas.

AT*SQA provides this syllabus as follows:

1. To training providers, to produce courseware and determine appropriate teaching methods.
2. To credential candidates, to prepare for the exam (as part of a training course or independently).
3. To the international software and systems engineering community, to advance the profession of software and systems testing, and as a basis for books and articles.

The AT*SQA may allow other entities to use this syllabus for other purposes, provided they seek and obtain prior written permission.

Trademarks

The following registered trademarks and service marks are used in this document:

- Windows , Microsoft, Azure, and Azure DevOps are registered trademarks of the Microsoft Corporation.
- Apple and iOS are registered trademarks of Apple Corporation.
- Postman is a registered trademark of Postman Inc.
- ReadyAPI, TestComplete, SOAPUI, and TestExecute are registered trademarks of SmartBear Software.
- ASTQB is a registered trademark of the American Software Testing Qualifications Board
- ISTQB is a registered trademark of the International Software Testing Qualifications Board.
- ChatGPT is a registered trademark for OpenAI Company.
- Blazemeter is a registered trademark for the Perforce Software Inc.
- JMeter is a trademark of Apache JMeter.

Works Cited

James Newton-King. (2022, 09 22). Compare gRPC services with HTTP APIs. Retrieved from https://learn.microsoft.com: https://learn.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-8.0

James Newton-King. (2022, 09 20). Test gRPC services with Postman or gRPCurl in ASP.NET Core. Retrieved from learn.microsoft.com: https://learn.microsoft.com/en-us/aspnet/core/grpc/test-tools?view=aspnetcore-7.0

Ana. (2023, 1 19). What are API Headers. Retrieved from mixedanalytics.com: https://mixedanalytics.com/knowledge-base/api-headers-explained/#:~:text=in%20API%20Connector%3F-,What%20are%20API%20headers%3F,an%20API%20key%20for%20authentication.

contributors, M. (2022, 9 9). Accept. Retrieved from developer.mozilla.org: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept

Esteban Herrera. (2022, 08 24). GraphQL vs. REST APIs: Why you shouldn't use GraphQL. Retrieved from https://blog.logrocket.com/: https://blog.logrocket.com/graphql-vs-rest-api-why-you-shouldnt-use-graphql/

Hitesh Baldaniya . (2021, 07 19). Why and When to Use GraphQL. Retrieved from https://dzone.com: https://dzone.com/articles/why-and-when-to-use-graphql-1

Jan Kratochvil. (2023, 01 13). A new Postman integration for Azure DevOps users. Retrieved from https://blog.postman.com/: https://blog.postman.com/postman-integration-for-azure-devops-users/

jwt.io. (n.d.). introduction. Retrieved from jwt.io: https://jwt.io/introduction

Levy, T. (n.d.). A Machine Learning Approach to Log Analytics. Retrieved from logz.io: https://logz.io/blog/machine-learning-log-analytics/

MDN contributors. (2022, 10 23). Content-Type. Retrieved from developer.mozilla.org/: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type

Microsoft. (n.d.). Data contract overview. Retrieved from https://learn.microsoft.com: https://learn.microsoft.com/en-us/industry/retail/intelligent-recommendations/data-contract

MIcrosoft. (n.d.). JavaScriptSerializer.MaxJsonLength Property. Retrieved from learn.microsoft.com: https://learn.microsoft.com/en-us/dotnet/api/system.web.script.serialization.javascriptserializer.maxjsonlength?view=netframework-4.8.1

Mozilla. (n.d.). HTTP response status codes. Retrieved from mozilla.org: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

mozilla.org. (n.d.). Synchronous and asynchronous requests. Retrieved from mozilla.org: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests

RedHat. (2020, 5 8). What is a REST API? Retrieved from www.redhat.com: https://www.redhat.com/en/topics/api/what-is-a-rest-api

Redhat.com. (2022, 6 2). What is an API? Retrieved from www.redhat.com: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces

RFC. (n.d.). RFC-2616. Retrieved from RFC-2616: https://www.rfc-editor.org/rfc/rfc2616#section-6

Tamura, K. (2015, 04 21). The log: The lifeblood of your data pipeline. Retrieved from www.oreilly.com: https://www.oreilly.com/content/the-log-the-lifeblood-of-your-data-pipeline/

Tesauro, M. (n.d.). API Security Tools. Retrieved from owasp.org: https://owasp.org/www-community/api_security_tools

W3 Schools. (n.d.). JSON Data Types. Retrieved from W3 Schools: https://www.w3schools.com/js/js_json_datatypes.asp

Tesauro, M. (n.d.). API Security Tools. Retrieved from owasp.org: https://owasp.org/www-community/api_security_tools

W3 Schools. (n.d.). JSON Data Types. Retrieved from W3 Schools: https://www.w3schools.com/js/js_json_datatypes.asp

AT*SQA

MICRO-CREDENTIAL

API Testing

REST API, gRPC, and graphQL

www.atsqa.org