Certified API, REST & Microservices Tester (CARMT) with Postman or Karate Syllabus

Version: 1.0 2021

Released: 1st of October, 2021

Thank you to the main author

- Dr. Srinivas Padmanabhuni

Thank you to the co-authors

- Gaurav Pandey, Jayapradeep Jiothis, José Díaz & Viepul Kocher

Thank you to the review committee

- Alexis Herrera, Alfonso Fernández, Ángel Rayo Acevedo, Christine Green, Danilo Ramirez, Émilie Potin-Suau, Enrique Alejandro Decoss Martinez, Fabiola Mero, Gastón Marichal, Guino Henostroza, Héctor Ruvalcaba, Isaac Marcelo Malamud Kobrinsky, Ismael Betancourt, Javier Chávez, Joan Tasayco, Julie Gardiner, Julio Córdoba Retana, Kyle Alexander Siemens, Leandro Melendez, Manu Eyckmans, Márcia Araújo Coelho, Márcia Campos, Mario Alvarez Gómez, Miaomiao Tang, Miguel Angel De León Trejo, Miroslav Renda, Neriman Kocaman, Oscar Alejandro Arreola Ramirez, Orlando Torres, Richard Seidl, Samuel Ouko, Thomas Cagley, Valeria Cocco, Vanessa Islas Padilla, Wilson Gumba & Wim Decoutere.

Revision History

Version	Date	Remarks
0.1 for Review Committee	Nov, 2020	Review Committee
0.2 for Review Committee	August, 2021	Final Prep for global release
1.0 Global Release	October, 2021	Global release

Table of Contents

Business Outcomes

BO-1	Understand the current trends and industry applications of application programming interfaces (APIs).
BO-2	Understand how to compare different API standards to help choose the most suitable one.
BO-3	Be able to compare APIs of REST and SOAP types.
BO-4	Understand how API testing complements UI Testing.
BO-5	Define a test strategy for testing of APIs.
BO-6	Understand how data-driven testing can be applied to API testing.
BO-7	Understand how testing of Microservices differs from API testing.
BO-8	Use API test tools to automate testing of APIs.

Learning Objectives/Cognitive Levels of Knowledge

Learning objectives (LOs) are brief statements that describe what you are expected to know after studying each chapter. The LOs are defined based on Bloom's modified taxonomy as follows:

- K1: Remember. Some of the action verbs are Remember, Recall, Choose, Define, Find, Match, Relate, Select
- K2: Understand. Some of the action verbs are Understand, Summarize, Generalize, Classify, Compare, Contrast, Demonstrate, Interpret, Rephrase
- K3: Apply. Some of the action verbs are Implement, Execute, Use, Apply

Hands-on Objectives

Hands-on Objectives (HOs) are brief statements that describe what you are expected to perform or execute to understand the practical aspect of Learning Objectives.

The HOs are defined as follows:

- HO-0: Live demo of an exercise or recorded video
- HO-1: Guided exercise. The trainees follow the sequence of steps performed by the trainer
- HO-2: Exercise with hints — Exercise to be solved by the trainees utilizing hints provided by the trainer
- HO-3: Unguided exercises without hints

Prerequisites

Mandatory

- None

Recommended

- ISTQB® Certified Tester Foundation Level
- Basic knowledge of JavaScript
- Basic knowledge of developing web-based applications
- Some software development or testing experience

Chapter 1 - Introduction to Application Programming Interfaces (API)

Keywords: API, Interfaces, REST, SOA, Microservices, Standards, Integration.

LO-1.1.1	K2	Understand what an application programming interface (API) is.
LO-1.1.2	K1	Recall the main characteristics of APIs: standards, implementation independence, client independence.
LO-1.2.1	K2	Understand the advantages of using APIs.
LO-1.2.2	K2	Explain how API-based integrations work.
LO-1.2.3	K2	Explain different flavors of APIs, in particular: REST, SOA and Microservices.
LO-1.2.4	K1	Recall the differences between SOA and Microservices.
LO-1.2.5	K1	Recall the differences between REST and SOA.

1.1 Application Programming Interface (API)

1.1.1 Definition of API

LO-1.1.1	K2	Understand what an application programming interface (API) is.

An API can be defined as:

- the interface of a software application to the outside world
- a standards-based interface enabling any generic IT application to talk to any other application

- an interface whose granularity can vary from interfacing a very small functional module to a complete application interface.

1.1.2 Main Characteristics of APIs

LO-1.1.2	K1	Recall the main characteristics of APIs: standards, implementation independence, client independence.

The main characteristics of APIs are:

- based on strict standards
- can be called by other applications over the internet, intranet and extranet
- allow applications to be connected by loosely coupled connections, one side application does not make any assumption about the application on the other side
- consuming client applications that communicate with the interface have to be developed independently of the underlying software application
- should allow the underlying implementation of the interface to be independent of any client software application communicating to the interface
- standards evolve through industry consensus

1.2 Advantages and Implementation of API Usage

1.2.1 Advantages of API Usage

LO-1.2.1	K2	Understand the advantages of API Usage.

Architectures based on APIs have the following advantages:

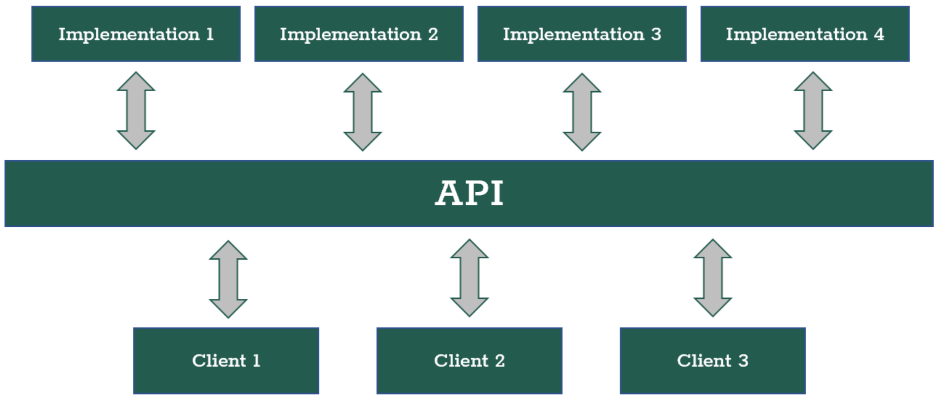
- platform-independent
    - eliminates the need for communicating applications to make platform choices like J2EE versus .NET versus mainframe
- independent of communicating applications
    - high reusability across current and future communicating applications
- legacy applications preserving

    - preserves existing IT investments by providing standard interfaces to existing applications
- warrant incremental investment
    - allows for incremental investment in technology, as opposed to fresh solutions, via the use of interfaces over existing applications
- are graphical user interface (GUI) agnostic
    - allows for multiple GUI technologies to be built on top of its core functionality
- reduced overall integration cost
    - because of standardized APIs, the cost of integration is lowered

1.2.2 Integration with API

LO-1.2.2	K2	Explain how API-based integrations work.

Due to the availability of standards-based interfaces on both the application and client sides, the overall integration cost of connecting and integrating multiple IT applications and IT clients is minimized. Previous to the emergence of APIs, the integration between IT applications and clients would depend upon costly middleware software.



Conventionally, M x N number of connectors for M applications to communicate with N client applications (spaghetti integration). In the current API centric integration, it would reduce to a M + N applications which would be much less than M x N for any positive values of M and N greater than zero. This reduces the integration complexity and cost.

1.2.3 Different Types of APIs

LO-1.2.3	K2	Explain different flavors of APIs, in particular: REST, SOA and Microservices.

REST is a term used to describe an architecture style of distributed systems utilized to connect resources using standard networks. The key idea in REST is to access remote objects over standard networks with http as the basic protocol, and uniquely addressing objects by using a unique identifier. A resource in REST could refer to a HTML page, an image, a file, or any object accessible over the internet. REST enables a standard way for clients to connect to applications over the internet, using http as the underlying protocol.

SOA describes an architecture style of distributed systems built on standards-based interfaces for IT applications. IT applications are called using standards-based interfaces termed as services. Services lower the cost of the integration of IT applications. There is an end-to-end standards stack for SOA. The SOA stack consists of an XML (See Ref [XML]) based set of standards for language, data, the messaging, and publishing of interfaces. Alongside, there is a set of standards for non-functional requirements like security, and reliability.

Microservices are a fine-grained API type, where an application is structured as a collection of small autonomous services. This architecture separates portions of a (usually monolithic) application into small, complete services.

1.2.4 Differences Between SOA and Microservices

LO-1.2.4	K1	Recall the differences between SOA and Microservices.

The differences between SOA and Microservices are listed below:

SOA	Microservices
• higher granularity • application wrapped up as a service • protocols are verbose • communication across heterogeneous systems • standards-based	• smaller granularity • application made up of Microservices • protocols are lightweight • communication within homogeneous systems • non-standard

1.2.5 Differences Between SOA and REST

LO-1.2.5	K1	Recall the differences between SOA and REST.

SOA	REST
• more standardized • allows one-way as well as two-way messaging • can work for http as well as other protocols • standards for all system architecture layers — messaging, interface description, security, data, etc. • uses XML and verbose models	• only protocol is standardized • allows only two-way messaging • protocol is http • non-standardized data, messaging structure, interface description • uses lightweight data formats

Chapter 2 - REST Architecture

Keywords: REST, http, GET, POST, DELETE, CRUD, PUT, HEAD, PATCH, URI, Query String

LO-2.1.1	K2	Explain the REST architecture characteristics.
LO-2.2.1	K1	Recall the different parts of REST interfaces.
LO-2.3.1	K2	Explain different REST methods - HEAD, GET, POST, DELETE, PUT and PATCH.
LO-2.3.2	K2	Explain different testing needs of REST interfaces.

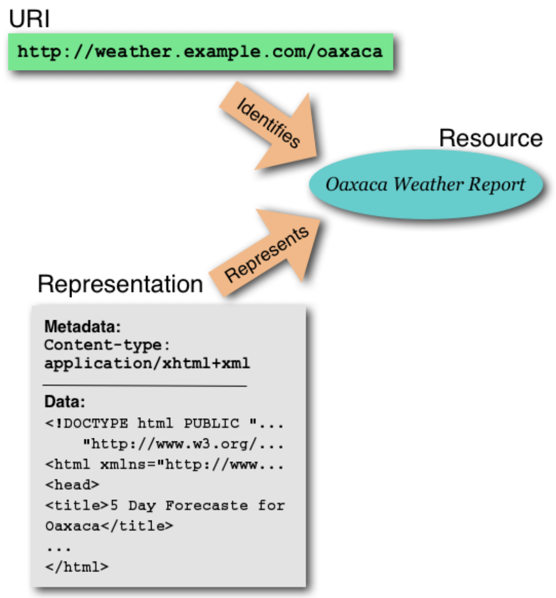
2.1 REST Architecture

2.1.1 Basics of REST Architecture

LO-2.1.1	K2	Explain the REST architecture characteristics.

REST is an acronym standing for Representational State Transfer, and every item of interest in REST is called a resource.

2.1.1.1 Uniform Resource Identifier (URI)

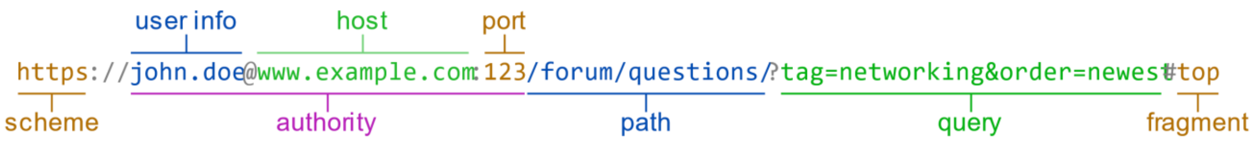


Each resource has a unique Uniform Resource Identifier (URI),as shown in the illustration.

A URI is a string which uniquely identifies a resource on the internet. URI can be of two types: Uniform Resource Locator (URL) or Uniform Resource Name (URN).

A URL is a subset of URI which additionally specifies the mechanism of obtaining the resource.

Structure of a URI (a URL):


user info
host
port
https://john.doe@www.example.com:123/forum/questions/?tag=networking&order=newest#top
scheme
authority
path
query
fragment

Source: Ref [url]

The different parts of a URI (specifically URL) are:

- Scheme: it indicates the protocol to access the resource, e.g. http/https/FTP/SMTP/File. (See Ref [url])
- Authority: section, preceded often by :// or :/// which consists of an optional username, mandatory hostname or IP address and an optional port number (the port number is not used when the resource is on a default port for a given protocol).
- Path: is a sequence of path segments (each separated by /) which indicates the location of the resource relative to the top domain. The path may be fixed or may involve variable values. For instance, the device ID can be part of the path where the ID will vary for different devices.

- Query String: the last key value pair of URL where there is a set of key value pairs, preceded by ? (e.g. ?k1=v1&k2=v2). An example from the search engine ABC for a search query:
  https://www.abc.com/search?q=Bangalore+Club&search=&form=QBLH
  - the scheme (protocol) is **https**
  - the authority is **www.abc.com** (port 443 is assumed due to it being the default port for https)
  - the path is **/search** (in some URLs, only path exists without any query string)
  - the Query String is **?q=Bangalore+Club&search=&form=QBLH**
- Fragment: is a relative location within the web page.

These parts of the URL collectively indicate accessing a function on the server, which uses the arguments provided in the query string.
A URN, on the other hand, is an unambiguous way to identify a resource.
Example: an ISBN number.
All URNs and URLs are URIs but not vice versa.
Example of URN : `urn:isbn: 0439064872`

2.1.1.2 Characteristics of REST Architecture

REST interfaces allow a web-based resource to be accessed from any other client application on a network. The resource could be an actual object on the server, or it could be a verb (a method) to be called on the server. Some of the most important characteristics of REST architecture are:

- every REST method is a client server style interaction, with a request and a corresponding response.
- REST methods typically use http(s) as the protocol for web-based access to REST resources.
- a REST method will be stateless by default, i.e., it means that no stored context on the server can be used.
- all resources can only be accessed by basic methods HEAD, GET, POST, DELETE, PUT and PATCH.
- the format of a resource URI is not standard, so that one server may put variables in the resource part, for example, "https://endpoint.com/devices/1", while the other may want to put it in the query string, for example, "https://endpoint.com/devices?deviceid=1". Both are acceptable designs.

2.2 The Main Components of REST

2.2.1 Components of REST

LO-2.2.1	K1	Recall the different parts of REST interfaces.

The REST interfaces contain different components, each of which are explored in subsequent sections:

- Request URI
- Request Headers
- Response Headers
- Response Codes
- Response Body

2.2.2 Request URI

HO-2.2.2	H2	Decompose some real URIs to identify different parts of the request.

A request URI is typically of the form:
https://TLD/Resource?k1=v1&k2=v2&k3=v3

Top Level Domain (TLD) represents the host name or the IP address. The resource can be a hierarchical name (e.g. search/b/c) or a simple name (e.g. search). The query string at the end is a collection of key value pairs, for example:

https://www.abcz.com/search?q=Bangalore+Club&search=&form=QBLH

- the TLD is **www.abcz.com**
- the resource is **search**
  - the Query String is **?q=Bangalore+Club&search=&form=QBLH**

The variable parts of the request URI are either in the resource part or in the query string or in both places.

2.2.3 Request Headers

HO-2.2.3	H2	Examine request headers in a typical REST Request.

These are the request headers that are commonly present in http requests:

- Accept: specify desired media type of response
- Accept-Language: specify desired language of response
- Date: date/time at which the message was originated
- Host: host and port number of a requested resource
- If-Match: conditional request
- Referrer: URI of previously visited resource
- User-Agent: identifier string for web browser or user agent. Based on this header, the server usually sends a customized response for different browsers. For example, a mobile app will be sent a smaller amount of content (only important content) as compared to a browser from a desktop.
- Cookie: matched cookies for state maintenance

2.2.4 Request Body

HO-2.2.3	H2	Examine request body in a typical REST Request.

A REST request has an optional request body component that is commonly present in methods other than GET. This body can contain detailed data, large documents which usually cannot be sent as part of a URL.

2.2.5 Response Headers

HO-2.2.5	H2	Examine response headers in a typical REST method call.

These are the commonly found response headers in the http response:

- Allow: list the REST methods supported by the request URI
- Content-Language: language of the response content
- Content-Type: media type of representation, where typical content types are text/html, application/json, etc.

- Content-Length: length in bytes of the representation
- Date: date/time at which the message was originated
- Expires: date/time after which the response is considered obsolete
- Last-Modified: date/time at which the representation was last changed
- Set-Cookie: sets cookies for the domain. This is useful for maintaining the state, as http basically is stateless.

2.2.6 Response Codes

HO-2.2.6	H2	Examine different response codes in multiple real-world REST method calls.

There are five categories of response codes. The response codes of http (Source: Ref [co1]) are summarized below:

- 1XX: Informational
- 2XX: Success
  200 OK, 201 Created, 202 Accepted, 204 No Content
- 3XX: Redirection
  300 Multiple Choices, 301 Moved Permanently, 302 Moved Temporarily, 304 Not Modified
- 4XX: Client Error
  400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found
- 5XX: Server Error
  500 Internal Server, Error 502 Not Implemented

2.2.7 Response Body

HO-2.2.7	H2	Examine the response body in a real-world REST method.

The response body is the content returned from the server in response to a REST method call. Typically, this response is in form of JSON. JSON stands for JavaScript Object Notation. This is a lightweight data format used for storing and transporting data. JSON is easy to understand and is self-explanatory.

An example of a JSON response is:
{"gpa":4.0,"gre":65,"id":2344,"rank":2}

This illustrates a JSON record with four keys and the corresponding values. This particular record represents a student's details. More complex JSON objects can be formed by nesting JSON records inside other JSON records.

2.3 The Different Methods of REST

2.3.1 Methods of REST

LO-2.3.1	K2	Explain different REST methods - HEAD, GET, POST, DELETE, PUT and PATCH.
LO-2.3.2	K2	Explain different testing needs of REST interfaces.

The main methods in REST architecture based on http(s) are:

- GET
- POST
- PUT
- PATCH
- DELETE
- HEAD

In the context of REST architecture, there are semantics analogous to the database operations (Create, Read, Update, Delete) for the key methods. This mapping provides a clear design principle for designing REST interfaces. The mapping can be roughly illustrated as below:

- GET — read operation
- POST — create records
- PUT/PATCH — update records
- DELETE — delete records
- HEAD — return header

2.3.2 GET Method

HO-2.3.2	H3	Perform the invocation of a GET REST request.

GET is the preferred method for accessing a resource on the internet. Hence it is equivalent to the Read operation in databases. Much like the SELECT clause in SQL, you request to return a set of data items. The format of a GET request varies in terms of the underlying URI. The URI format is dependent upon the application provider. The important characteristics of the GET request are outlined below. A GET request:

- is a read-only request
- does not modify any resource
- is idempotent, i.e., when you request the same resource twice, it should give the same response
- puts all the content in the URI
- typically, does not put anything in the body
- URI can take at best 64 KB of data
- is not encouraged when there is a need to send secured data.

2.3.3 POST Method

HO-2.3.3	H3	Perform invocation of a POST REST request.

POST is usually used when there is a need to send content of more than 64KB, even for a database read operation. The important characteristics of the POST request are listed below. A POST request:

- is used for sending secured data
- uses simple URIs but sends most content in the request body part
- as it typically corresponds to the database operation of creating a record, it is recommended when there is a need to create a new resource on the server
- is not idempotent

2.3.4 PUT Method

HO-2.3.4	H3	Perform invocation of a PUT REST request.

as it typically corresponds to the database operation of updating a record, PUT method is recommended when there is a need to update an existing resource on the server. The important characteristics of the PUT request are listed below. A PUT request:

- is idempotent
- may use content in URI or in BODY
- updates the entire record, so all the field values (updated) should be provided

2.3.5 PATCH Method

HO-2.3.5	H3	Perform invocation of a PATCH REST request.

PATCH is the same as PUT, except that you can do a partial update of the resource on the server, unlike PUT, where all the fields are updated. As a result, you can update one or two fields as well.

2.3.6 DELETE Method

HO-2.3.6	H3	Perform invocation of a DELETE REST request.

The DELETE method is used to indicate to the server to delete a stated resource.

2.3.7 HEAD Method

HO-2.3.6	H3	Perform invocation of a HEAD REST request.

The HEAD method is a special case of the GET request, where only the response headers are returned with no body. It is usually used to test response codes, or to test if the server is up and running.

Chapter 3 – SOA

Keywords: SOA, Web Services, SOAP, WSDL, XML Schema,

LO-3.1.1	K1	Define SOA.
LO-3.1.2	K2	Understand the standards stack of SOA.
LO-3.1.3	K2	Summarize the features of XML schemas.
LO-3.1.4	K2	Summarize the features of SOAP.
LO-3.1.5	K2	Summarize the features of WSDL.

3.1 Basics of Service-Oriented Architecture (SOA)

3.1.1 Defining SOA

LO-3.1.1	K1	Defining Service-Oriented Architecture (SOA).

SOA is an approach to software development that takes advantage of reusable software applications, or services. It is an architecture style of distributed systems based on standards-based interfaces for IT applications. IT applications are published, located, and called using standards-based interfaces termed as services. Services lower the cost of the integration of applications, due to standards-based interfaces. Different companies and enterprises across the globe have come together to define an end-to-end standards stack for SOA. The SOA standards stack consists of an XML-based set of standards for language, data messaging and the publishing of interfaces. There is also a set of standards for non-functional requirements like security and reliability.

3.1.2 Defining the Standards Stack for SOA

LO-3.1.2	K2	Understand the standards stack of SOA.

SOA is based on a universal set of standards for each functioning layer of distributed systems. In the context of the functioning of distributed systems, there is a whole set of standards for all the functional and non-functional layers. The concise stack is presented in the figure below.

3.1.3 Defining the XML Schema

LO-3.1.3	K2	Summarize the features of XML schemas.

Usually, when a C client invokes a Java application, the number of bytes in C is different from that in Java. These types of incompatibility causes are primarily responsible for high integration costs. This data incompatibility can be overcome by having a universal standard for data. That data standard is an XML schema (see Ref [XMS]).

Any programming language has several data types for modeling data. An XML schema defines the standard for data types that are independent of the underlying language. Integration issues in data types usually happen when there is a heterogeneity of the implementations of the data types in the different languages.

An XML schema describes the grammar and structure of an XML document and defines the:

- elements in the document
- attributes of elements in the document
- order of child elements in the document
- data types of the elements in the document
- default and fixed values of attributes and elements

There are two types of data types — Basic and Complex:

- Basic data types are the common data types generally found in programming languages, e.g., Int, String, Float, etc.
- Complex data types are custom data types created out of basic data types or other complex data types. An example of complex data type:

In this data, the schema talks of a product information structure, where there are two basic elements: prodname is a text string, and barcode is an integer.

3.1.4 Defining SOAP

LO-3.1.4	K2	Summarize the features of SOAP.

HO-3.1.4	H3	Perform invocation of a SOAP request.

SOAP (see Ref[SOP]) is a universal standard for messaging, independent of the underlying protocol being used. Several integration scenarios are possible in a typical enterprise integration, that consists of both one-way and two-way messaging. SOAP is modeled as a one-way messaging standard, independent of the underlying transport layer of the network. A SOAP message can be sent over a one-way protocol like SMTP, or over a two-way protocol like http(s).

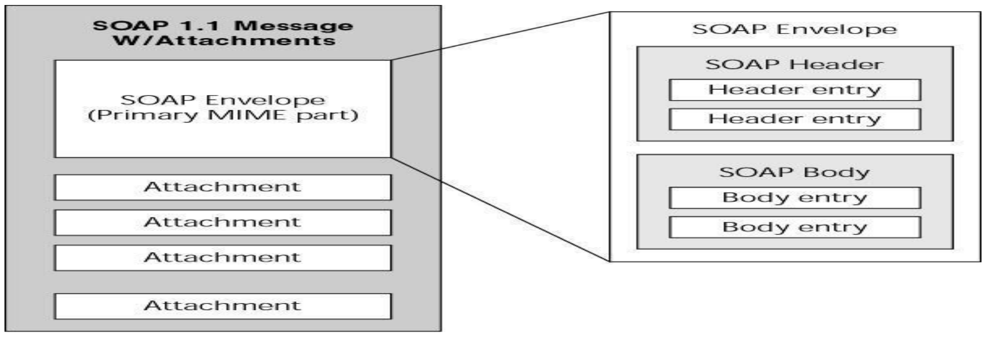
SOAP defines the

- syntax and semantics for representing data in SOAP messages
- communication model for exchanging SOAP messages
- bindings (conventions and implementations) for specific transport protocols like SMTP, or http
- conventions for both a two-way remote procedural call and a one-way messaging

In any protocol, SOAP represents a one-way message. In SMTP, it will be a one-way alert SOAP message. In a http binding, there are two SOAP messages, one for SOAP request and another for SOAP response.

A typical SOAP message as above consists of three parts: a SOAP Envelope, a SOAP Header (optional) and a SOAP Body. In addition, each SOAP message can have a set of attachments to the message.

SOAP Message
(Ref [SOF])



SOAP Envelope is the overall wrapper for the SOAP message. It consists of an optional SOAP header and the actual SOAP body.

SOAP Header consists of zero or more header entries, each of which usually consists of non-functional requirements related to the SOAP message transmissions, e.g. security-related headers, billing-related headers, and user identification headers, etc.

A SOAP Envelope has a single child SOAP Body node. A SOAP Body node contains one or more body entries. The body entry could be a RPC method and its parameters, or a custom XML message sent to the destination. When there is an error in a http SOAP request, the relevant message is wrapped and sent as SOAPFault.

Example of a typical SOAP-based http request:

```
POST /BookPrice http/1.1
Host: catalog.acmeco.com
Content-Type: text/xml;charset="utf-8" Content-Length: 640
SOAPAction:"GetBookPrice"

<SOAP-ENV:Envelope>
 xmlns:SOAP ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3c.org/2001/XMISchema-instance"
 xmlns:xsd="http://www.w3c.org/2001/XMISchema"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Header>
  <person:mail xmlns:person="http://acmeco.com/Header/">
   xyz@acmeco.com
  </person:mail>
 </SOAP-ENV:Header>
 <SOAP-ENV:Body>
  <m:GetBookPrice xmlns:m="http://www.wiley.com/jws.book.priceList">
   <bookname xsi:type='xsd:string'>
    Developing Java Web Services
   </bookname>
  </m:GetBookPrice>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example of a typical SOAP-based http response:

```
http/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 640

<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
```

```
        xmlns:xsi="http://www.w3c.org/2001/XMLSchema"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <SOAP-ENV:Header>
         <wiley:Transaction
          xmlns:wiley="http://jws.wiley.com/2002/booktx"
          SOAP-ENV:mustUnderstand="1">
          5
         </wiley:Transaction>
        </SOAP-ENV:Header>
        <SOAP-ENV:Body>
         <m:GetBookPriceResponse
          xml:ns="http://www.wiley.com/jws.book.priceList">
          <Price>
           50.00
          </Price>
         </m:GetBookPriceREsponse>
        </SOAP-ENV:Body>
       </SOAP-ENV:Envelope>
```

This example shows that the transport binding of SOAP involves a request and a response. The data in SOAP messages clearly include data in form of an XML schema.

3.1.5 Defining WSDL

LO-3.1.5	K2	Summarize the features of WSDL.
HO-3.1.5	H3	Load a WSDL in SoapUI or a related tool.

WSDL (see Ref[WSD]) is an XML language used to *describe* and *locate* web services. The different components of WSDL give detail about the functionality of a service, and the location of the service implementation. WSDL provides all the details required to enable any client to connect to the service and invoke the service.

There are two parts of a WSDL: abstract WSDL and physical WSDL.

The abstract WSDL, known as a portType, contains the details of the functionality of the service, independent of the language and location. The physical WSDL, known as binding, consists of the physical implementation of the interface, including port, IP address and transport details. A typical WSDL example:

```
       <definition namespace = “http/… “>
              <types> xschema types </types>
```

```
      <message> … </message>
      <porttype> a set of operations
            <operation> set of messages either input/output or only input or
only output </operation>
            </porttype>
      <binding> communication protocols </binding>
      <service> a list of binding and ports </service>
<definition>
```

The portype node contains the abstract WSDL part, and the service node contains the physical WSDL part. An end-to-end example can be seen via the link at Ref [WS1].
The node `<types>` defines types used in a message declaration in the form of an XML schema:

```
<types>
   <schema targetNamespace="http://example.com/stockquote.xsd"
xmlns="http://www.w3.org/2000/10/XMLSchema">
        <element name="TradePriceRequest">
             <complexType>
                   <all>
                       <element name="tickerSymbol" type="string“
                          minOccur = “1” maxOccur=“10”>
                       <element name = “payment”>
                             <complexType> <choice>
                                  <element name = “account” type=“string”>
                                  <element name = “creditcard” type=“string”>
                             </choice> </complexType>
                        </element>
                   </all>
             </complexType>
        </element>
   </schema>
</types>
```

Here, the `<types>` tag is indicating an element that corresponds to a request for getting the current stock price of a list of stocks. The example indicates that it has two elements: a tickerSymbol which can take 1 to 10 stock symbols, and a payment tag, which can be either an account or a credit card (both strings).

The `<message>` element defines the data elements of an operation. Each message can consist of one or more parts. The parts can be compared to the parameters of a function call in programming.

```
<message name="GetLastTradePriceInput">
      <part name="body" element="TradePriceRequest"/>
</message>
<message name="GetLastTradePriceOutput">
```

```
        <part name="body" element="TradePrice"/>
</message>
```

The `<portType>` element is the most important WSDL element. It defines the functionality of **a web service**, the **operations** that can be performed, and the **messages** that are involved. Each **operation** actually represents the kind of message exchange allowed by the web service.

```
<portType name="StockQuotePortType">
   <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
   </operation>
</portType>
```

This example indicates a request-response kind of operation, which takes stock names, and returns current prices of the stocks. WSDL defines four operation types, request-response being the most common operation type:

- **One-way**: The operation can receive a message but will not return a response
- **Request-response**: The operation can receive a request and will return a response
- **Solicit-response**: The operation can send a request and will wait for a response
- **Notification**: The operation can send a message but will not wait for a response

Example of one-way request and response:

```
<portType name=“RegisterPort">
   <operation name=“register">
      <input name=“customer Info" message=“RegInfo"/>
   </operation>
   <operation name = “register Response”>
      <output name = “response” message=“ResponseInfo”/>
   </operation>
</portType >
```

Binding defines how messages are transmitted, and the location of the service, for example:

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
     <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
     <operation name="GetLastTradePrice">
          <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
```

```
                <input>
                    <soap:body use="literal"/>
                </input>
                <output>
                    <soap:body use="literal"/>
                </output>
           </operation>
      </binding>
```

The final node is *service*, which is a collection of bindings, along with actual address for clients to invoke, for example:

```
      <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
             <soap:address location="http://example.com/stockquote"/>
        </port>
      </service>
```

Chapter 4 - Microservices

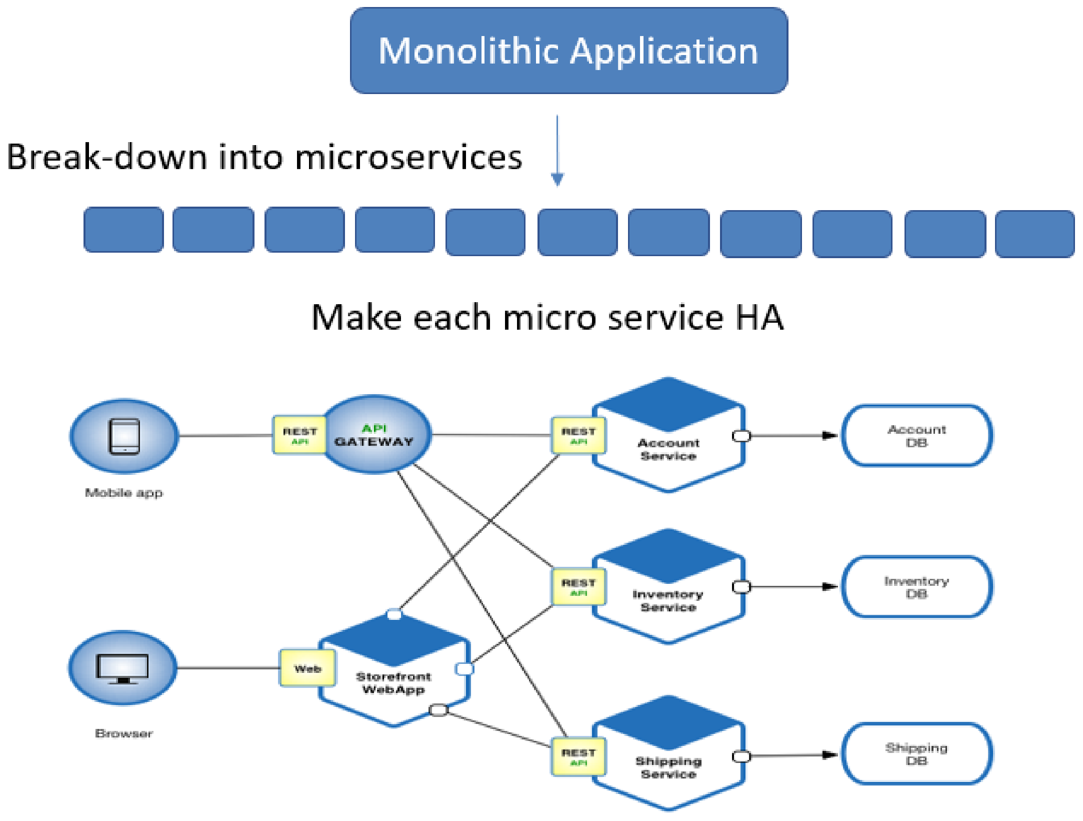
Keywords: Microservices, REST, serverless, containers

4.1 Basics of Microservices

LO-4.1	K1	Define what Microservices are.
HO-4.1	H1	Examine the interface of a Microservice.

Microservices are basically services, however, they are of much smaller granularity. An application is made up of several Microservices. The protocols are lightweight, and communication is within homogeneous systems. The APIs are typically based on REST and non-standard. Figure 4.1 illustrates the typical Microservices design pattern.

Fig. 4.1 Typical Microservice Implementation (Ref [AW1])



This diagram shows a typical Microservices architecture. A monolithic application is usually converted to several fine grained Microservices. Any Microservice has the following specific characteristics:

Fine Grained: it is usually very fine grained, e.g., a single method, and at the level of application components where several Microservices are integrated to form an application.

Autonomous: From birth, to deployment, to management, the Microservice must manage itself in its lifetime without depending upon other Microservices.

Resilient: Microservices architecture must show resiliency, i.e., the application should continue to run independently of the failure of a specific Microservice.

Self-Contained: since the life cycle of the Microservice is maintained by itself, without depending upon other Microservices, the Microservice should be self-contained. As seen in Fig. 4.1, the Microservice maintains its own database and application logic.

Chapter 5 - Testing APIs

Keywords: test design techniques, positive testing, negative testing, security testing, equivalence partitioning, boundary value analysis, coverages, resiliency testing, mocks

LO-5.1.1	K1	Recall the level of API testing in the test pyramid.
LO-5.1.2	K1	Recall the advantages of API testing over GUI testing.
LO-5.2.1	K3	Apply the positive testing identification techniques to API testing.
LO-5.3.1	K3	Apply the negative testing identification techniques to API testing.
LO-5.4.1	K1	Understand the security needs of API testing.
LO-5.4.2	K3	Apply security tests when testing APIs.
LO-5.5.1	K3	Apply load tests when testing APIs.
LO-5.6.1	K3	Apply orchestration tests when testing APIs.
LO-5.7.2	K3	Apply different kinds of assertions in API testing.
LO-5.8	K1	Understand the need for mocking APIs.
LO-5.9	K1	Understand the specific testing needs of Microservices.

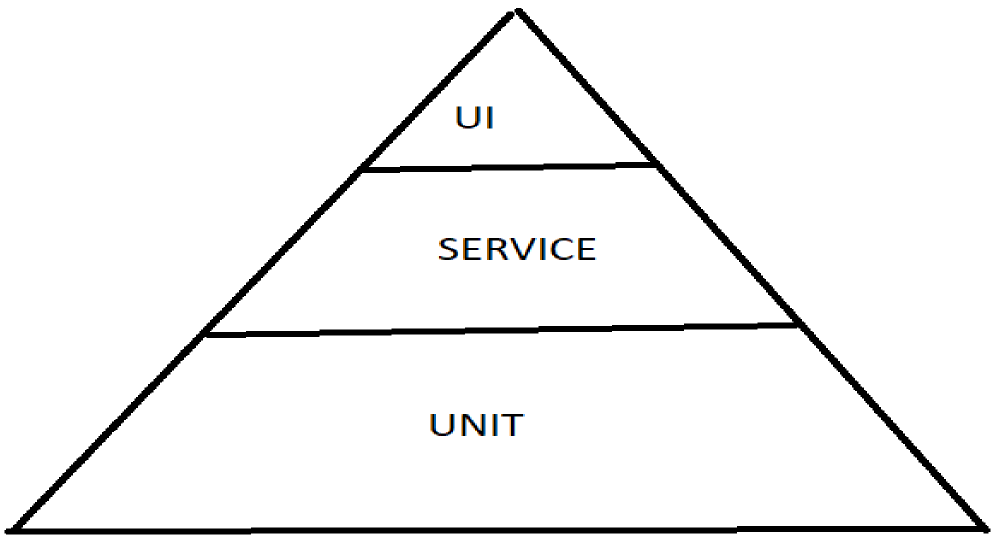
5.1 API Testing in the Test Pyramid

5.1.1 API Testing in the Test Pyramid

LO-5.1.1	K1	Recall the level of API testing in the test pyramid.

In a typical test pyramid, API testing, also referred to as service testing, is below the UI testing, and above unit testing. It can be viewed at the level of component integration testing. Typically, several varieties of UIs can be built using reusable APIs, which offer reusable data and methods to be passed on the UI layer. In general, APIs are of higher granularity than UIs.

Figure 5.1.1 Importance of API testing in the test pyramid



(Source: Ref[FO1])

5.1.2 Advantages of API Testing vs. GUI Testing

LO-5.1.2	K1	Recall the advantages of API testing over GUI testing.

The different advantages of API testing are:

- a shift left in software testing life cycle because of the frequent reuse of the methods and resources in APIs, across UIs where there is an early detection of any functional bugs before they reach the GUI stage. In the development process, the functionality in API is designed, developed, and tested before GUI, hence shift left.
- the APIs describe stable contracts and do not change too often, leading to lower maintenance costs of APIs
- the API layer offers a reusable resource used in multiple APIs, the actual logic inside the APIs is tested before the GUI layer.
- the concise interfaces, which describe the APIs, render the testing of APIs less time consuming.
- language independent due to XML/JSON as content — a key benefit of APIs is the language independence due to the use of XML or JSON, which reduces the number and variety of test cases to be tested.

API testing generally requires less time and is a more efficient way to test in comparison to GUI testing.

5.2 Positive Testing of APIs

5.2.1 Positive Testing Techniques

LO-5.2.1	K3	Apply the positive testing techniques to API testing.

5.2.1.1 Test Techniques for Testing of APIs

HO-5.2.1.1	H4	Apply typical test techniques to a real-world API.

The usual test techniques can be applied to real-world APIs, including REST and SOAP. Some of the typical techniques applicable in this case would be:

- Equivalence Partitioning
- Boundary Value Analysis
- Exploratory Testing
- State Transition Testing
- Decision Table Testing

- Classification Tree Testing
- Pairwise Testing

5.2.1.2 Application of the Positive Testing of APIs

HO-5.2.1.2	H4	Apply typical positive test cases after deriving them in a real-world API including response codes, content inspection tests, schema compliance tests, etc.

Positive testing is a type of software testing that is performed by assuming everything will be as expected. Positive testing is not limited to payload and body, but it also applies to headers and parameters in REST as well as in SOA. If positive test cases fail, this shows its implementation was not complete.

Typical positive test cases for APIs involve the following tests from a functional testing perspective:

- testing for headers in response
- testing for headers in request
- testing for right response codes
- testing of valid values in fields in resource
- testing of values in fields in parameters
- testing of API response bodies compliance to JSON schema/XML schema
- testing of response content parts in JSON or XML body or plain text
- testing of pre-request scripts

Typical positive test cases for APIs involve the following tests from a non-functional testing perspective:

- testing of proper response times
- testing of authentication of APIs
- testing of authorization of APIs
- testing of graceful degradation of APIs
- testing of performance of APIs
- testing of edge cases for performance of APIs
- testing of privacy of APIs
- testing of multiple authentication factors in accessing the APIs

5.3 Negative Testing of APIs

5.3.1 Application of Negative Testing of APIs

LO-5.3.1	K3	Apply the negative testing identification techniques to API testing.
HO-5.3.1	H4	Apply typical negative test cases after deriving them in a real-world API, including illegal values, illegal types, out of bounds values, etc.

Negative testing is a type of software testing that is performed by feeding wrong data to the APIs and examining the behavior of the API. If negative tests fail, it indicates improper exception handling. Due to the incomplete negative testing of APIs, they fail in the real world when malafide data is fed to the APIs.

Typical negative test cases are listed below:

- field values of illegal type (not of allowed type) in input
- field values out of bounds (allowed range) in input
- field values not allowed in input
- field value is null while it is mandatory
- field is omitted
- required field is excluded
- duplicate fields
- invalid XML or JSON not conforming to schema
- empty request
- invalid request header (an important header)
- in security, sending insecure data
- in security, sending wrong credentials
- in database APIs, sending corrupt data
- sending wrong METHOD (PUT instead of GET)
- no data in body when it expects data
- testing for multiple requests for load

5.4 Security Testing of APIs

5.4.1 Security Requirements of APIs

LO-5.4.1	K1	Understand the security needs of API testing.

As per the latest ISO 25010 standard (Ref [IS1]) security testing is a non-functional testing type and it is one of the crucial quality characteristics. Typical IT applications work on security to achieve the following objectives towards the application. These applications are collectively known as the CIA triad (Source Ref[CIA]).

Confidentiality (C): refers to the indication that the persons who are not authorized to access an IT asset are not given access to it. This is achieved usually by encryption.
Integrity (I): refers to ensuring that the information is not tampered on the way to its destination, and it remains in its original form. This is usually achieved by checksums and hash codes.
Availability (A): refers to making sure that the relevant information is made available to genuine users.

A variety of techniques exist to guarantee the CIA triad for IT applications security. The availability of APIs is guaranteed by a combination of authentication and authorization techniques. The differences between authentication and authorization is:

- Authentication: refers to establishing the identity of a user accessing a system with a view to allowing only genuine users to be given access.
- Authorization: refers to mechanisms used to establish access control levels for resources to users.

Testing for security is an important facet of the API testing life cycle. The top 10 API security concerns as per OWASP (Ref [OWS]) range from broken authentication to different broken authorization systems.

5.4.2 Different Security Tests of APIs

LO-5.4.2	K3	Apply security tests when testing APIs.

HO-5.4.2	H4	Perform security testing to real APIs, including different authentication types.

The typical security concerns, as highlighted in the previous section, provide the need for different kinds of security tests designed for APIs. Some of the security tests specifically designed for APIs include:

Test for authentication: different kinds of authentication mechanisms have been designed for the authentication needs of APIs. According to the type of authentication, the corresponding test case should use the technique to do both positive and negative testing with correct as well as incorrect users. The different kinds of authentications which are used in APIs include:

- username/password basic authentication: the username and password are sent as plaintext after encoding the string as a base64 encoded form.
- digest authentication: the password is not sent, but a checksum computed from the password is sent, and is checked at server.
- bearer authentication: a token is expected to be sent from client to the server, and access is granted only if the request carries this token
- Kerberos authentication: over an insecure channel, a client proves its identity to the server with a ticket.

Test for authorization: in the authorization of APIs, there is a need to test for different access control mechanisms. Test for authorization involves checking for the appropriate access control mechanisms via both positive and negative tests.

Testing for encryption: depending upon the confidentiality need for the API, the API might use different levels and kinds of encryption mechanisms for handling the confidentiality of the API.

Penetration testing: with the objective of preventing potential threats to the APIs, a comprehensive penetration testing of APIs is to be performed with the right analysis of threats, and vulnerabilities, and testing with the appropriate test cases. The vulnerabilities of APIs are described in the OWASP top ten (Ref[OWS]).
Fuzz testing: It includes fuzzing the API parameters to unreasonable values to test the API response and hence bring out any errors in the API backend.

5.5 Load Testing of APIs

LO-5.5.1	K3	Apply load tests when testing APIs.
HO-5.5.1	H3	Perform load testing to real APIs including different load types.

Load Testing is an important non-functional testing addressing the performance of systems (Ref [IS1]). Since APIs are exposed to different kinds of application data in their invocations, and are usually called over the Internet, it is vital that their availability is ensured for all genuine API users. To achieve availability for all API users, it is important to test the API in terms of its performance with respect to different load patterns. Several tools exist to simulate different load patterns for the APIs, like JMeter, LoadUI, K6, etc.

These tools typically help in testing how an API can handle an expected number of concurrent requests. They do it by artificially simulating calls for load simultaneously. Being able to create simultaneous requests, they measure metrics like response time, throughput, connection time, overall time.

The different patterns of API load tests include:

- Load test: it tests by accessing the API via the creation of several concurrent virtual users. Response times are measured and checked to find out whether they correspond to the expected numbers.
- Soak test (Endurance test): this kind of test measures the endurance of the system by sending normal load for a long duration of time.
- Spike test: this is testing for sudden spikes in the load by causing sudden peaks and observing the responses to them.

5.6 Integration Testing of APIs

LO-5.6.1	K3	Apply orchestration tests when testing APIs.
HO-5.6.1.1	H3	Perform sequence testing of different APIs.

HO-5.6.1.2	H3	Perform the data driven orchestration testing of different APIs.
HO-5.6.1.3	H3	Perform state-based orchestration testing using the global and environment variables of different APIs.
HO-5.6.1.4	H3	Perform real test oracle implementation by comparing with with expected outputs.

Since APIs are standalone calls to applications delivering some functionality, very often, more complex real-life scenarios involve combining multiple APIs in a workflow to achieve the desired output. Such tests involving multiple APIs is termed as orchestration testing.

The orchestration tests for API testing can involve several patterns:

1. A simple hard coded sequence of APIs.
2. Context sensitive orchestrations for APIs, where data at intermediate stage dictates the subsequent APIs to be called.
3. Data-driven API orchestrations — different instances with varied arguments are used to execute the same API, with arguments being in various components of the API, be it the body or the URI, or the header. The API testing tools allow for the parametrization of these parts of a URI or header or body, and these parameterized data are fed into the tool from an external CSV file or text file with the data.
4. Data Driven Test Oracles — if the expected outputs in response to data driven API orchestrations are also fed in the tool from an external CSV file, along with the values of the parameters, then a data driven test oracle can be emulated in API testing orchestrations.

Some of the typical test design techniques are listed below:

- Decision Table testing
- Decision Tree testing
- Pairwise testing

5.7 Automation in API Testing

5.7.1 Assertions and Automation

Since APIs are effectively interfaces to functions, automating testing is possible in APIs, just like any normal testing. The automation is achieved by specifying the expected outputs and comparing this to the actual output of the API called. In the context of API testing, this automation has to be enabled, all the while taking into consideration the fact that the outputs could be in different components of the APIs (the response codes, the response body)

These specifications are termed as Assertions, and they play an important role in the automation of API testing. There are several kinds of assertions in the context of API testing. Some of the important categories of these assertions are listed in the following sections.

5.7.2 Different Assertions in API testing

LO-5.7.2	K3	Apply different kinds of assertions in API testing.
HO-5.7.2.1	H3	Perform test automation with different assertions for different APIs in tools like SoapUI and postman.

Generally, assertions are applied on the actual content output of an API, REST or SOAP. The range of assertions could involve a range of patterns:

- Whole Text Contains: just checking an expected text in the whole response.
- JSON/XML node data inspection: the most challenging of the content inspections, this includes a systematic exploration of the JSON or XML documents and comparing a specific subnode to match an expected value. JSONPath or XPath based assertions are common in this kind of assertion. API Testers may need to traverse through a whole array of the data in the response, to find specific matching JSON/XML entries which can then be examined for further content comparison.

- Header Inspection: typically associated with response or request headers, these assertions test the presence or absence of headers, and in some cases, the exact match of the values of the headers with expected values.
- Response Code Assertions: this kind of assertions are more common in REST and SOAP, in both cases to examine the http response code with an expected response code. A detailed set of response codes has been explained in Section 2.2.6. In some cases, it is expected that an improper access code is returned, such as 404. In that case, the appropriate negative testing assertion should be used.
- Schema Compliance Assertions: in both JSON and XML, the documents usually follow a well-defined grammar in the form of the underlying JSON Schema or XML Schema. These assertions test for the compliance of the responses or requests with respect to the underlying XML Schema or JSON Schema grammars.
- Response Time SLA Assertions: typically associated with asserting expected response time, these assertions will flag the SLA adherence to or violations of the SLA response time asserted.
- Interoperability Assertions: specifically, in web services, there is a WS-I interoperability profile adherence related assertion.

5.8 Mocking of Services

LO-5.8	K1	Understand the need for mocking APIs.
HO-5.8.1	H3	Perform the mocking of different APIs and test the mock services.

Typically, a copy of the original API is used in mocking, in place of the original API. The different use cases of mocking are:

- Simulation when implementation is not ready, specifically like complex APIs
- Other services need to use the API, but the implementation is not ready for integration testing
- Performance testing of APIs on a simulated version of the original implementation, to avoid any issues in the original implementation behind the API

- Some APIs cannot be tested in production
- You have to pay to use another API

Several forms of mocking have been proposed in APIs. Some of these are:
- Mocks
- Stubs
- Service virtualization

5.9 Testing Microservices

LO-5.9	K1	Understand the specific testing needs of Microservices.

In most cases, the APIs of Microservices are exposed as REST interfaces. Therefore, all the tests required for API testing in REST are necessary for Microservices. These include the functional tests, positive tests, negative tests for REST, and non-functional tests for REST APIs. There are two special cases of Microservices which need a specific kind of testing in the case of Microservices:
- Contract First Testing
- Resiliency Testing

5.8.1 Contract First Testing

LO-5.8.1	K2	Understand the steps in the contract first testing of Microservices

Within an application, there is a strong need for efficient integration between the different Microservices. The integration of two Microservices requires that we test the interactions between the two microservices, as part of the application.

A contract is technically the set of interactions between two Microservices. This leads to a strong contract first testing of the contract between the provider Microservice and consumer Microservice. The consumer Microservice should test for the contract adherence of the provider Microservice. Likewise, the provider Microservice should intimate any changes in the contract to all the consuming Microservices to enable efficient integration testing. In several cases, in which the consumer Microservice and provider Microservice are developed at the same time, testers need to maintain the synchronization between the two Microservices. In this context, special tools are used to maintain this synchronization. These tools test a consumer Microservice by mocking the provider Microservice, then by testing the provider Microservice with a mock of the consumer Microservice. The tools known as "brokers" keep tab of the contracts, and use the contracts as the basis for testing.

5.8.2 Resiliency Testing

LO-5.8.2	K2	Understand the Resiliency testing of Microservices

Resiliency generally refers to the capability of an IT application to recover from failures gracefully. Since Microservices are heavily dependent upon each other to integrate into the application, the overall Microservices ecosystem needs to be very resilient. In that context, it is important to run effective simulations of different failure scenarios of the network. Tools for the resiliency testing of Microservices involve generating scenarios like randomly bringing down Microservices, and cutting off some connections between different Microservices randomly.

Glossary

API: Application Programming Interface is the interface of a software application through which the software application is called.

Granularity: Size of a software program; it can be of small, limited functionality, or large functionality.

Client-Server: A kind of distributed systems architecture with two kinds of systems: the providers of a resource, termed as servers, and resource requesters termed as clients.

SOA: SOA is a special kind of client server architecture where software applications are accessed using standards-based interfaces.

Microservices: A special kind of client server architecture where software applications of small granularity are made accessible to external software clients.

Protocol: A set of rules defining the transmission of data between two nodes in a computer network.

Loosely Coupled: A style in which a calling software application does not make any assumption about the called software application and vice versa.

XML: Stands for eXtensible Markup Language, which is a standard structured format for data, to be read by humans or machines.

Web Services: A set of standards for implementing SOA.

GUI: Stands for Graphical User Interface, a set of visual components for interacting with the software.

SOAP: Stands for Simple Object Access Protocol, which refers to a Web service standard for messages exchanged over a network.

WSDL: Stands for Web Service Description Language, which refers to a Web service standard used to describe the details of the interface of a service.

XML Schema: A Web service standard used to describe a universal grammar for data, independent of programming language, based on XML.

Autonomous: Refers to the concept of self-managing software.

Java: An object-oriented programming language.

C: A procedural programming language.

JavaScript: A programming language for the web.

JSON: Stands for JavaScript object notation, which is a format used to represent data in a structured hierarchical format.

URI: Stands for Uniform Resource Identifier which uniquely identifies any resource on the Internet.

URL: A special form of URI, stands for Uniform Resource Locator, and is used to identify network resources on the Internet.

URN: A special form of URI, stands for a generic approach to naming an object, different from URL.

HTTP: Stands for Hypertext Transfer Protocol, a special kind of computer network protocol for clients used to send and receive messages to/from a special type of server: Web server.

Contract: Stands for the set of interactions between an IT provider and an IT consumer.

Confidentiality: Information is not made available or disclosed to unauthorized individuals, entities or processes.

Integrity: safeguarding the accuracy and completeness of an information asset.

Availability: being accessible and usable upon demand by an authorized entity.

Authentication: ensuring the only genuine people access an information asset.

Authorization: ensuring that there are sufficient permissions to carry out an operation or to access an information asset.

Assertion: A statement expected to be true.

Mock: A replica of the original API.

Response Code: A three-digit code returned by an HTTP request.

Resiliency: Capability of recovering from a failure gracefully.

Invocation: Calling a function or an API.

Orchestration: Coordinating several APIs in a sequence or workflow.

References

[ur] https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml

[ur1]
https://upload.wikimedia.org/wikipedia/commons/5/51/URI_Components_Full_Example_httpS.svg

[co1] https://datatracker.ietf.org/doc/html/rfc7231

[WS1] http://www.dneonline.com/calculator.asmx?wsdl

[FO1] https://martinfowler.com/bliki/TestPyramid.html

[CIA] https://en.wikipedia.org/wiki/Information_security

[OWS] https://www.netsparker.com/blog/web-security/owasp-api-security-top-10/

[XML] https://www.w3.org/XML/

[XMS] https://www.w3.org/XML/Schema

[SOP] http://www.w3.org/TR/SOAP

[WSD] https://www.w3.org/TR/wsdl.html

[SOF] https://docs.oracle.com/cd/E19159-01/819-3669/6n5sg7br6/index.html

[AW1] AWS Lambda Documentation

[IS1] https://www.iso.org/standard/35733.html