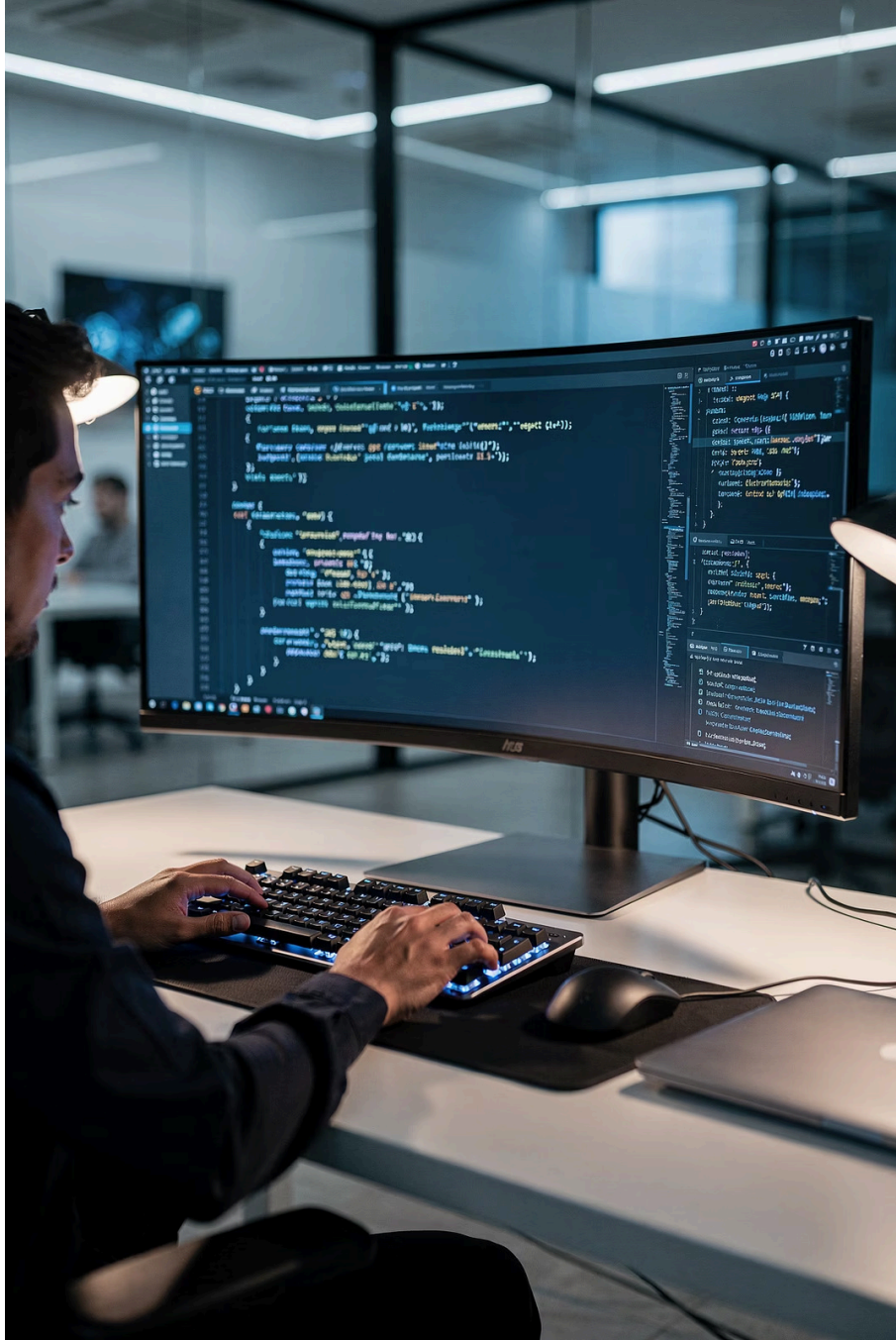
API Testing Fundamentals – REST, SOAP & SoapUI Hands-On

A comprehensive training programme designed for QA engineers, automation testers, and manual testers transitioning to API testing. This course provides practical, hands-on experience with REST and SOAP protocols using industry-standard tools.



What is an API?

An **Application Programming Interface (API)** is a software intermediary that allows two applications to communicate with each other. APIs define the methods and data structures that developers use to interact with external software components, operating systems, or microservices.

Think of an API as a waiter in a restaurant: you (the client) tell the waiter (the API) what you want, the waiter communicates your order to the kitchen (the server), and then brings back your food (the response).

Request

Client sends data with specific parameters

Processing

Server executes business logic

Response

Server returns result to client

Why APIs Are Critical in Modern Applications

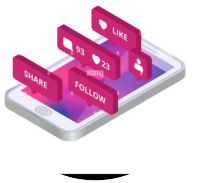
APIs have become the backbone of modern software architecture, enabling seamless integration between systems, platforms, and services. They power everything from mobile banking to social media feeds.

Banking & Finance

Payment gateways, account balance checks, fund transfers, and third-party integrations all rely on secure API communication.

E-Commerce

Product catalogues, inventory management, order processing, and payment systems communicate through REST and SOAP APIs.



Mobile Applications

Mobile apps consume APIs to fetch data, authenticate users, and synchronise information across devices in real-time.

API Testing vs UI Testing

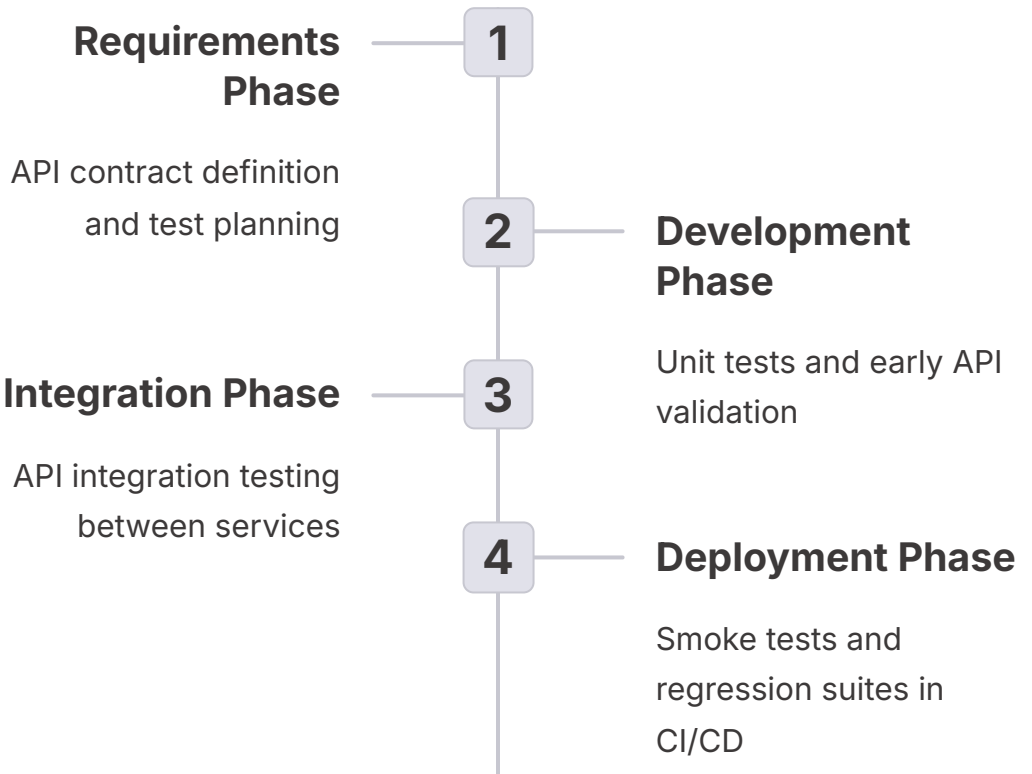
Understanding the distinction between API testing and UI testing is crucial for comprehensive test coverage. Both testing types serve different purposes and complement each other in a robust QE strategy.

Aspect	API Testing	UI Testing
Test Layer	Business logic layer	Presentation layer
Speed	Fast execution	Slower due to rendering
Maintenance	Less brittle, stable	High maintenance cost
Coverage	Tests data and logic	Tests user workflows
Tools	SoapUI, Postman, REST Assured	Selenium, Cypress, Playwright
Error Detection	Early in development cycle	Later in development cycle

Where API Testing Fits in SDLC

API testing plays a critical role throughout the Software Development Lifecycle (SDLC), particularly in Agile and DevOps environments. It enables early defect detection, faster feedback loops, and supports continuous integration and deployment practices.

Modern QE practices emphasise **shift-left testing**, where API tests are executed as soon as the service contract is defined, even before the UI is developed. This approach reduces the cost of finding and fixing defects significantly.



Knowledge Check: API Fundamentals

Question 1

Which statement best describes an API?

A) A user interface for end users
B) A contract that defines how software components communicate
C) A database management system
D) A programming language

Answer: B – An API is a contract or interface that defines methods and data structures for communication between software components.

Question 2

What is the primary advantage of API testing over UI testing?

A) Better visual validation
B) Faster execution and early defect detection
C) No technical knowledge required
D) Only tests the frontend

Answer: B – API testing executes faster, doesn't depend on UI changes, and enables early testing in the development cycle.

INTERVIEW KEYWORDS

API • Request • Response • Integration • Business Logic Layer • Shift-Left Testing

Client–Server Communication

Client–server communication forms the foundation of how APIs work. The **client** is any application that requests data or services, whilst the **server** is the system that provides those resources. This communication follows a request-response pattern over HTTP/HTTPS protocols.

1. **Client**
   Initiates request with required parameters
2. **Network**
   Transmits request over HTTP/HTTPS
3. **Server**
   Processes request and generates response
4. **Client**
   Receives and processes response data

Request and Response Structure

HTTP Request Components

Request Line

Contains HTTP method, URI, and protocol version

Headers

Metadata like Content-Type, Authorization, Accept

Body

Payload data (for POST, PUT, PATCH requests)

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer token123

{
  "name": "John Smith",
  "email": "john@example.com"
}
```

HTTP Response Components

Status Line

Protocol version, status code, and reason phrase

Headers

Response metadata like Content-Type, Cache-Control

Body

Response payload (JSON, XML, HTML, etc.)

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/users/12345

{
 "id": 12345,
 "name": "John Smith",
 "created": "2024-01-15T10:30:00Z"
}
```

Key API Terminology

Endpoint

The specific URL where an API can be accessed. Example: `https://api.example.com/v1/users`

Resource

An object or data entity that the API exposes. Example: users, products, orders

Payload

The actual data sent in the request body or received in the response body

Headers

Metadata sent with requests/responses containing information like content type and authentication tokens

Parameters

Additional information sent with requests: query parameters (?id=123), path parameters (/users/123), or body parameters

Status Codes

Three-digit numbers indicating the result of the request (2xx success, 4xx client errors, 5xx server errors)

HTTP Status Codes Overview

HTTP status codes are standardised responses that indicate the outcome of an API request. Understanding these codes is essential for effective API testing and debugging.

Code	Category	Meaning & Examples
2xx	Success	**200 OK** – Request succeeded **201 Created** – New resource created **204 No Content** – Success but no response body
3xx	Redirection	**301 Moved Permanently** – Resource moved to new URI **304 Not Modified** – Cached version is still valid
4xx	Client Errors	**400 Bad Request** – Invalid request syntax **401 Unauthorised** – Authentication required **404 Not Found** – Resource doesn't exist
5xx	Server Errors	**500 Internal Server Error** – Generic server failure **503 Service Unavailable** – Server temporarily down

Knowledge Check: API Concepts

Question 1

Which component of an HTTP request contains authentication credentials?

A) Request body
B) Query parameters
C) Request headers
D) Status line

Answer: C – Authentication tokens are typically sent in request headers, commonly using the Authorization header field.

Question 2

What does a 404 status code indicate?

A) Server error
B) Resource not found
C) Unauthorised access
D) Successful response

Answer: B – HTTP 404 indicates that the requested resource could not be found on the server. It's a client error status code.

What Are HTTP Methods?

HTTP methods (also called HTTP verbs) define the type of action to be performed on a resource. They are fundamental to RESTful API design and indicate the intended operation: retrieving data, creating new resources, updating existing ones, or deleting them.

Understanding HTTP methods is crucial for API testing, as each method has specific characteristics, expected behaviours, and appropriate use cases. Proper validation requires knowing which method should be used for each operation.

Role of HTTP Methods in REST APIs

CRUD Operations

HTTP methods map directly to CRUD (Create, Read, Update, Delete) operations in RESTful services. This standard mapping ensures consistency and predictability across APIs.

01

Create

POST method

02

Read

GET method

03

Update

PUT or PATCH

04

Delete

DELETE method

Method Properties

Each HTTP method has specific properties that determine how it should behave:

- **Safe methods** – Do not modify server state (GET, HEAD, OPTIONS)
- **Idempotent methods** – Multiple identical requests have the same effect as a single request (GET, PUT, DELETE)
- **Non-idempotent methods** – Multiple requests may produce different results (POST, PATCH)

These properties are critical for understanding expected API behaviour during testing and for designing reliable distributed systems.

GET Method

Purpose & Characteristics

The **GET** method retrieves data from the server without modifying any resources. It is the most commonly used HTTP method and should only be used for read operations.

Safe

Does not change server state

Idempotent

Multiple calls return same result

Cacheable

Responses can be cached

Real-World Usage

- Fetching user profile information
- Retrieving product catalogue or search results
- Loading order history or transaction details
- Getting current account balance

Example Request

```
GET /api/users/12345 HTTP/1.1
Host: example.com
Accept: application/json
```

Testing Considerations

Verify that GET requests never modify data, return appropriate status codes (200, 404), include correct response payload, and handle query parameters properly.

POST Method

Purpose & Characteristics

The **POST** method submits data to create a new resource on the server. Each POST request typically creates a new entity, making it non-idempotent – multiple identical requests create multiple resources.

Not Safe

Modifies server state

Non-Idempotent

Multiple calls create multiple resources

Returns 201

Success returns 201 Created with location header

Real-World Usage

- Creating new user accounts or registrations
- Placing orders or processing payments
- Submitting feedback or creating support tickets
- Uploading files or documents

Example Request

```
POST /api/orders HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "userId": 12345,
  "productId": 789,
  "quantity": 2
}
```

Testing Considerations

Validate that new resources are created with unique IDs, verify 201 status with Location header, check duplicate submissions, and ensure proper error handling for invalid data.

PUT Method

Purpose & Characteristics

The **PUT** method replaces an entire resource with new data. It is idempotent, meaning multiple identical requests produce the same result. If the resource doesn't exist, PUT can create it (though this is implementation-dependent).

Not Safe

Modifies server state

Idempotent

Multiple identical calls have same effect

Full Replace

Replaces entire resource

Real-World Usage

- Updating complete user profile information
- Replacing product details in inventory
- Updating configuration settings
- Modifying shipping address for an order

Example Request

```
PUT /api/users/12345 HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "id": 12345,
  "name": "Jane Smith",
  "email": "jane@example.com",
  "role": "admin"
}
```

Testing Considerations

Verify complete resource replacement, ensure omitted fields are removed or set to defaults, confirm idempotency by making multiple identical requests, and validate 200 or 204 status codes.

PATCH Method

Purpose & Characteristics

The **PATCH** method applies partial updates to a resource, modifying only specified fields. Unlike PUT, PATCH doesn't replace the entire resource. The idempotency of PATCH depends on implementation.

Not Safe

Modifies server state

May Be Idempotent

Depends on implementation

Partial Update

Updates only specified fields

Real-World Usage

- Updating only email address in user profile
- Changing order status from pending to shipped
- Modifying specific product attributes like price
- Updating account preferences or settings

Example Request

```
PATCH /api/users/12345 HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "email": "newemail@example.com"
}
```

Testing Considerations

Confirm only specified fields are modified, verify unspecified fields remain unchanged, test with invalid field names, and ensure appropriate error handling for partial updates.

DELETE Method

Purpose & Characteristics

The **DELETE** method removes a resource from the server. It is idempotent – deleting an already deleted resource should return the same result (typically 404) as the second deletion attempt.

Not Safe

Modifies server state

Idempotent

Multiple calls have same effect

Returns 204

Success typically returns 204 No Content

Real-World Usage

- Deleting user accounts or profiles
- Removing items from shopping cart
- Cancelling orders or subscriptions
- Removing uploaded files or documents

Example Request

```
DELETE /api/users/12345 HTTP/1.1
Host: example.com
Authorization: Bearer token123
```

Testing Considerations

Verify resource is actually deleted and cannot be retrieved, confirm appropriate status codes (200, 204, 404), test deletion of non-existent resources, and validate authorisation requirements before deletion.

HTTP Methods Comparison

Understanding the characteristics of each HTTP method is essential for both API design and testing. This comparison highlights the key properties that determine when and how to use each method.

Property	GET	POST	PUT	PATCH	DELETE
Safe	Yes	No	No	No	No
Idempotent	Yes	No	Yes	Sometimes	Yes
Request Body	No	Yes	Yes	Yes	No
Cacheable	Yes	Rarely	No	No	No
Success Code	200	201	200/204	200	204
Update Type	N/A	Create	Full Replace	Partial	Remove

Idempotent vs Non-Idempotent Methods

Idempotent Methods

An **idempotent** operation produces the same result regardless of how many times it is executed. Making the same request multiple times has the same effect as making it once.

GET

Reading data multiple times doesn't change the resource

PUT

Replacing a resource multiple times with same data results in same state

DELETE

Deleting a deleted resource still results in resource not existing

Non-Idempotent Methods

A **non-idempotent** operation may produce different results when executed multiple times. Each request may create a new resource or change the system state differently.

POST

Each request typically creates a new resource with a new ID

PATCH (sometimes)

Depending on implementation, may produce different results

Testing Implication: For idempotent methods, verify that multiple identical requests produce the same result. For non-idempotent methods, verify that each request is properly processed independently.

Knowledge Check: HTTP Methods

Question 1

Which HTTP method is idempotent but not safe?

A) GET
B) POST
C) PUT
D) PATCH

Answer: C – PUT is idempotent (multiple calls have same effect) but not safe (it modifies server state). GET is both safe and idempotent.

Question 2

What is the key difference between PUT and PATCH?

A) PUT is idempotent, PATCH is not
B) PUT replaces the entire resource, PATCH updates specific fields
C) PUT creates resources, PATCH deletes them
D) PUT requires authentication, PATCH does not

Answer: B – PUT performs a full replacement of the resource, whilst PATCH applies partial updates to specific fields only.

Question 3

If you call POST /api/orders three times with identical data, what is the expected outcome?

A) One order is created
B) Three orders are created
C) An error is returned
D) The request is cached

Answer: B – POST is non-idempotent, so each request typically creates a new resource, resulting in three separate orders with different IDs.

INTERVIEW TRAP

Common Interview Question: "Explain the difference between PUT and PATCH with examples." Be prepared to discuss full replacement vs partial update, idempotency considerations, and when to use each method.

What is REST?

REST (Representational State Transfer) is an architectural style for designing networked applications. It was introduced by Roy Fielding in his doctoral dissertation in 2000. REST is not a protocol or standard, but rather a set of constraints that, when applied, create a scalable, stateless, and cacheable communication system.

RESTful APIs use HTTP methods explicitly and are built around resources (nouns) rather than actions (verbs). Each resource is identified by a URI, and clients interact with these resources using standard HTTP methods.

83%

Of Public APIs Use REST

REST has become the dominant architectural style for web APIs

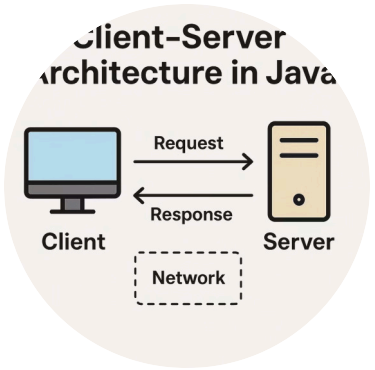
6

Core Constraints

Six architectural constraints define REST principles

Principles of REST Architecture

REST architecture is defined by six fundamental constraints. Understanding these principles is essential for designing, testing, and evaluating RESTful services effectively.



Client–Server Separation

Client and server are independent. The client handles the user interface whilst the server manages data storage and business logic.

Statelessness

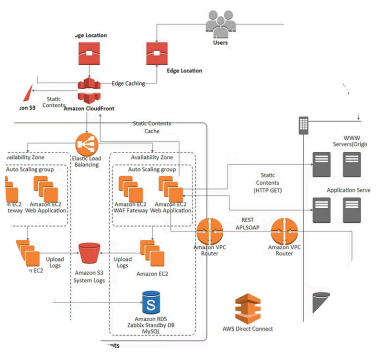
Each request contains all information needed to process it. The server doesn't store client context between requests.

Cacheability

Responses must define themselves as cacheable or non-cacheable to improve performance and scalability.

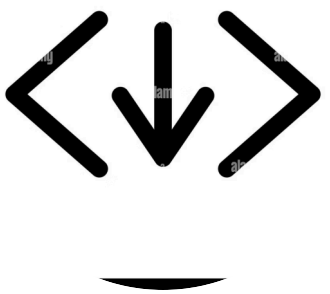
Uniform Interface

Resources are identified by URIs. Clients interact with resources through standardised representations.



Layered System

Architecture may be composed of hierarchical layers, with each layer only aware of the immediate layer it interacts with.



Code on Demand (Optional)

Servers can extend client functionality by transferring executable code, though this constraint is optional.

Statelessness in REST

Statelessness is one of the most important constraints in REST architecture. It means that each request from a client must contain all the information necessary for the server to understand and process the request. The server does not store any client context between requests.

Benefits of Statelessness

- **Scalability** – Servers can handle requests from any client without maintaining session state
- **Reliability** – Failure of a server doesn't affect client sessions
- **Visibility** – Each request can be understood in isolation
- **Simplicity** – Server implementation is simpler without session management

How Statelessness Works

1. Client includes authentication token in every request
2. Server validates token and processes request independently
3. Server sends complete response without storing session
4. Next request repeats the cycle with full context

Testing Implication: Verify that each API request is self-contained and doesn't depend on previous requests. Test authentication tokens are included and validated properly.

Resource-Based Architecture

REST APIs are organised around **resources** rather than actions. Resources represent entities in your domain model, such as users, products, or orders. Each resource is identified by a unique URI and can have multiple representations (JSON, XML, etc.).

Resource Naming Conventions

- Use nouns, not verbs in URIs
- Use plural names for collections
- Use lowercase and hyphens for readability
- Maintain hierarchical relationships

Good Examples

```
/api/users
/api/users/12345
/api/users/12345/orders
/api/products
/api/orders/789/items
```

Poor Examples (Anti-Patterns)

Bad URI	Why It's Wrong
/api/getUser	Uses verb instead of noun
/api/user	Singular for collection
/api/User	Uses uppercase
/api/users/delete	Action in URI (use DELETE method)

Testing Focus: Verify that resource URIs follow REST conventions, resources are properly nested for relationships, and HTTP methods are used correctly for each operation.

REST vs SOAP: High-Level Comparison

REST and SOAP are two different approaches to web services. Understanding their differences helps in selecting the appropriate technology for testing and in answering interview questions about architectural trade-offs.

Aspect	REST	SOAP
Type	Architectural style	Protocol with strict standards
Data Format	JSON, XML, HTML, plain text	XML only
Transport	HTTP/HTTPS	HTTP, SMTP, TCP, JMS
Message Size	Smaller, lightweight	Larger due to XML envelope
Performance	Faster, less overhead	Slower, more processing required
Security	HTTPS, OAuth, JWT tokens	WS-Security, built-in standards
State	Stateless	Can be stateful
Error Handling	HTTP status codes	SOAP fault elements
Contract	Optional (OpenAPI/Swagger)	Mandatory (WSDL)
Use Cases	Public APIs, mobile apps, microservices	Enterprise applications, financial services, legacy systems

Knowledge Check: REST Architecture

Question 1

Which of the following is NOT a constraint of REST architecture?

A) Statelessness
B) Client-Server separation
C) XML-only responses
D) Uniform interface

Answer: C – REST supports multiple data formats (JSON, XML, HTML, etc.). XML-only responses are characteristic of SOAP, not REST.

Question 2

What does "stateless" mean in REST architecture?

A) The server never changes its state
B) Each request contains all necessary information to process it
C) The client cannot maintain state
D) Resources cannot be modified

Answer: B – Statelessness means each request must be self-contained with all information needed. The server doesn't store client context between requests.

INTERVIEW KEYWORDS

Stateless • Resource • URI • Representation • Client-Server • Cacheable • Uniform Interface

What to Validate in REST APIs

Effective REST API testing requires comprehensive validation across multiple dimensions. Test coverage should span functional correctness, data integrity, performance, and security aspects.

Status Codes

Verify correct HTTP status codes for success, client errors, and server errors

Response Body

Validate JSON/XML structure, data types, required fields, and business logic

Response Headers

Check Content-Type, caching headers, CORS headers, and security headers

Data Correctness

Ensure data accuracy, consistency, and correct calculations or transformations

Error Handling

Test invalid inputs, missing required fields, unauthorised access, and edge cases

Performance

Measure response times, throughput, and behaviour under load conditions

Functional vs Non-Functional API Testing

Functional Testing

Validates that the API behaves according to specifications and business requirements.

Request Validation

Verify API accepts valid requests and rejects invalid ones

Response Validation

Check response structure, data types, and values

Business Logic

Ensure calculations, workflows, and rules execute correctly

Error Scenarios

Test error handling for invalid data and edge cases

Non-Functional Testing

Evaluates quality attributes like performance, security, and reliability.

Performance

Response time, throughput, latency under various loads

Security

Authentication, authorisation, data encryption, vulnerability testing

Reliability

Uptime, fault tolerance, recovery from failures

Scalability

Behaviour under increasing load, resource utilisation

Sample REST API Test Scenarios

These practical test scenarios demonstrate common validation patterns for REST APIs. Use these as templates when designing test cases for your projects.

1

Happy Path – Create User

Endpoint: POST /api/users
Input: Valid user JSON with all required fields
Expected: 201 Created, Location header with new user URI, response body contains user ID and creation timestamp

2

Validation Error – Missing Required Field

Endpoint: POST /api/users
Input: User JSON missing required email field
Expected: 400 Bad Request, error message specifying missing field

3

Authentication – Unauthorised Access

Endpoint: GET /api/users/12345
Input: Request without authentication token
Expected: 401 Unauthorised, appropriate error message

4

Resource Not Found

Endpoint: GET /api/users/99999
Input: Request for non-existent user ID
Expected: 404 Not Found, descriptive error message

5

Idempotency – Multiple PUT Requests

Endpoint: PUT /api/users/12345
Input: Send identical update request three times
Expected: Same result for all three requests, verify resource state remains consistent

Knowledge Check: REST API Testing

Question 1

Which status code should be returned when a POST request successfully creates a new resource?

A) 200 OK
B) 201 Created
C) 204 No Content
D) 202 Accepted

Answer: B – HTTP 201 Created is the correct status code for successful resource creation. It should include a Location header pointing to the new resource.

Question 2

What is the difference between functional and non-functional API testing?

A) Functional tests APIs, non-functional tests UI
B) Functional validates behaviour, non-functional validates quality attributes
C) Functional is manual, non-functional is automated
D) There is no difference

Answer: B – Functional testing verifies API behaviour against requirements, whilst non-functional testing evaluates performance, security, reliability, and other quality attributes.

What is SOAP?

SOAP (Simple Object Access Protocol) is a protocol for exchanging structured information in web services using XML. Unlike REST (which is an architectural style), SOAP is a formal protocol with strict standards defined by W3C.

SOAP was designed for enterprise applications requiring high security, ACID-compliant transactions, and formal contracts. It provides built-in error handling, supports multiple transport protocols, and includes comprehensive security features through WS-Security standards.

Protocol-Based

Formal standards and specifications

XML Only

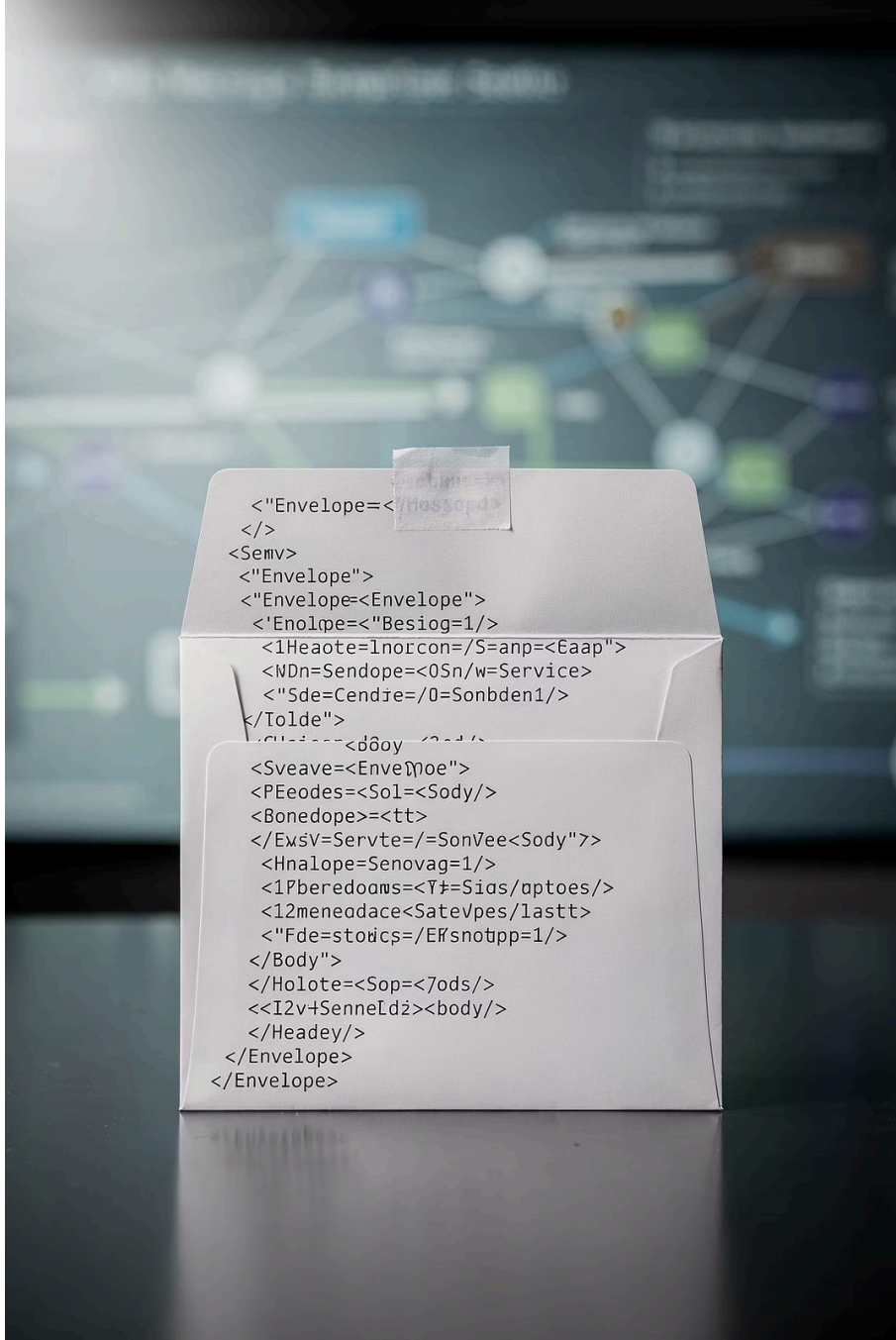
Messages use XML format exclusively

Transport Agnostic

Works over HTTP, SMTP, TCP, JMS

Enterprise Focus

Built for complex business requirements



SOAP Message Structure

Every SOAP message follows a standardised XML structure with specific elements. Understanding this structure is essential for testing SOAP services and debugging issues.

SOAP Components

01

Envelope

Root element identifying the document as a SOAP message

02

Header (Optional)

Contains metadata like authentication, transaction info

03

Body

Contains the actual message data and operation details

04

Fault (Optional)

Used for error messages if the request fails

Sample SOAP Message

```
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header>
  <auth:Token>abc123</auth:Token>
 </soap:Header>
 <soap:Body>
  <m:GetUserDetails>
   <m:UserId>12345</m:UserId>
  </m:GetUserDetails>
 </soap:Body>
</soap:Envelope>
```

Testing Focus: Verify envelope structure is valid, required namespaces are present, body contains correct operation, and fault elements appear for errors.

SOAP vs REST Comparison

Choosing between SOAP and REST depends on your specific requirements. Both have strengths and are appropriate for different scenarios.

Factor	SOAP	REST
Best For	Enterprise apps, financial services, legacy systems requiring high security	Public APIs, mobile apps, microservices, cloud services
Learning Curve	Steeper – requires understanding of XML, WSDL, namespaces	Gentler – uses familiar HTTP and simpler data formats
Standards	WS-Security, WS-AtomicTransaction, WS-ReliableMessaging	No formal standards – relies on HTTP, OAuth, JWT
Tooling	SoapUI, SOAP clients, enterprise tools	Postman, cURL, REST Assured, browser tools
Testing Complexity	More complex due to XML parsing and WSDL contracts	Simpler with JSON and flexible validation
Error Details	Structured fault elements with detailed error info	HTTP status codes with optional error body

Knowledge Check: SOAP Basics

Question 1

Which element is mandatory in every SOAP message?

A) Header
B) Fault
C) Envelope and Body
D) Authentication

Answer: C – Every SOAP message must have an Envelope element (root) and a Body element (containing the message data). Header and Fault are optional.

Question 2

What format does SOAP use for messages?

A) JSON
B) XML only
C) Plain text
D) Any format

Answer: B – SOAP exclusively uses XML format for messages. This is one key difference from REST, which supports multiple formats including JSON.

What is WSDL?

WSDL (Web Services Description Language) is an XML-based language used to describe the functionality offered by a SOAP web service. It serves as a contract between the service provider and consumer, specifying available operations, message formats, data types, and communication protocols.

WSDL enables automatic client code generation and provides complete documentation of the service interface. When you import a WSDL into SoapUI, the tool automatically generates request templates based on the service definition.

1

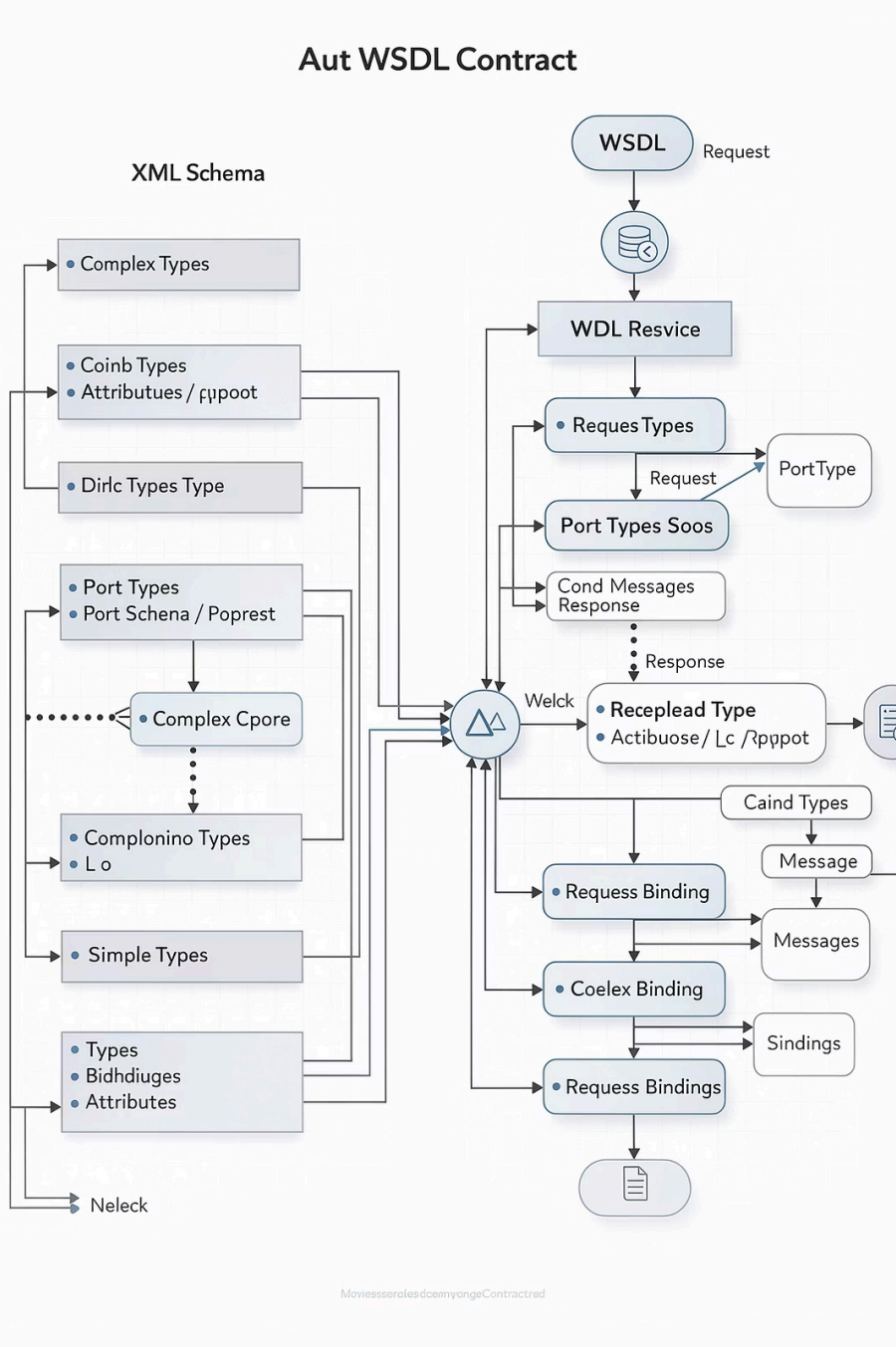
WSDL File

Completely describes a web service interface

7

Core Elements

Main components that define the service



WSDL Structure & Elements

A WSDL document contains several key elements that work together to fully describe a web service. Understanding these elements helps in testing and working with SOAP services.

1

Definitions

Root element containing all other elements. Defines namespaces and the target namespace for the service.

2

Types

Defines data types used by the service using XML Schema. Describes the structure of input and output messages.

3

Messages

Defines message formats for requests and responses. Each message contains one or more part elements referencing data types.

4

PortType

Defines abstract operations supported by the service. Similar to an interface in programming – specifies what operations are available.

5

Binding

Specifies concrete protocol and data format for operations defined in PortType. Defines how messages are transmitted.

6

Service

Defines the endpoint URL where the service is available. Contains one or more port elements that specify binding and location.

Why WSDL is Required

WSDL serves as a formal contract in SOAP-based web services, providing several critical benefits for both service providers and consumers.

Contract-First Development

WSDL enables a **contract-first** approach where the service interface is defined before implementation. This ensures clear agreements between teams and reduces integration issues.

Automatic Code Generation

Development tools can automatically generate client code from WSDL, reducing errors and accelerating development. Testers can import WSDL into SoapUI to instantly create test requests.

Service Documentation

Complete specification of operations, inputs, outputs

Interoperability

Enables different platforms to communicate reliably

Validation

Provides schema for validating requests and responses

Versioning

Manages service versions and backward compatibility

Knowledge Check: WSDL

Question 1

What is the primary purpose of WSDL?

A) To execute SOAP requests
B) To describe the interface and operations of a web service
C) To provide security for web services
D) To compress XML messages

Answer: B – WSDL describes the complete interface of a web service including operations, message formats, data types, and endpoints.

Question 2

Which WSDL element defines the actual endpoint URL where the service can be accessed?

A) PortType
B) Binding
C) Message
D) Service

Answer: D – The Service element contains port definitions that specify the binding and the actual endpoint URL (location) where the service is accessible.

INTERVIEW KEYWORDS

Contract-First • Service Definition • WSDL • PortType • Binding • Endpoint • Code Generation

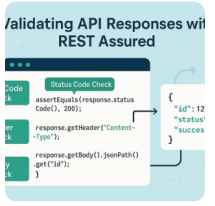
What is SoapUI?

SoapUI is a leading open-source tool for testing SOAP and REST web services. It provides a comprehensive platform for functional testing, load testing, and security testing of APIs without requiring programming knowledge.

SoapUI is widely used in enterprise environments due to its ability to import WSDL contracts, automatically generate test requests, validate responses against schemas, and create complex test suites with assertions and data-driven testing capabilities.

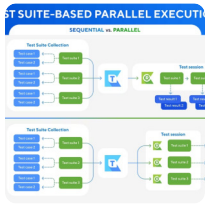
SOAP Testing

Import WSDL, generate requests automatically, validate XML responses



REST Testing

Test REST endpoints, work with JSON/XML, validate status codes and headers



Test Automation

Create test suites, add assertions, use Groovy scripts for advanced scenarios