IJC

International Journal of Computer (IJC)
Peer-Reviewed International Journal

ISSN: 2307-4523
https://ijcjournal.org

Comparative Analysis of API Testing Approaches in Service-Oriented Architectures

Aleksei Taranenko [a*]
[a]QA Automation Engineer, Phuket, Thailand
[a]Email: taacoder@gmail.com
ORCID 0009-0004-9252-2910

Accepted: 20 July 2026 | Published: 30 August 2026

Abstract

This article compares three approaches to automated application programming interface (API) testing for service-oriented and microservice architectures, where application programming interfaces serve as the principal points of interaction between independent services. The growing number of such systems underscores the need for a well-founded choice of approach, since each one supports protocols and test design styles differently. The objective is to qualitatively compare a fluent assertion syntax, a typed-client harness, and an action-based scenario engine, with a focus on test design, readability, maintainability, and scope of application. The three approaches are not interchangeable products, and the comparison is drawn between the test code each of them produces rather than between feature lists. The novelty lies in a direct comparison of reference implementations that solve one common scenario, reproduced in full in the text and evaluated against criteria tied to indicators observable in the code. The analysis shows that a flexible domain-specific syntax suits the verification of Representational State Transfer (REST) interfaces, that a typed client concentrates the contract in a single declaration, and that the multi-protocol engine covers integration scenarios and message-based interaction. The choice depends on the type of interaction and the maintenance horizon of the test base. The work will be useful for test automation engineers, quality architects, and software quality assurance instructors.

Keywords: API testing; service-oriented architecture; microservices; interaction contract; REST Assured; Retrofit; Citrus; test automation; integration testing; qualitative comparison

1. Introduction

Modern software systems are increasingly built as collections of independent services that exchange data via application programming interfaces (APIs), and this organization shifts a substantial part of the risk from the internal logic of a single module to the boundaries of service interaction [1].

Under such a design, the application programming interface becomes a contract whose violations cause failures that surface during the joint operation of services, and interface verification becomes a distinct testing level with its own tools and techniques. Engineers responsible for automation face a wide set of tools, each with their own model for describing a request, its own set of supported protocols, and its own style of expressing assertions. The choice among them is made only rarely through a systematic comparison of their properties.

The cost of an incorrect choice accumulates over the lifetime of a project. A tool that fits the

first service of a system may prove inadequate as the system grows toward heterogeneous interaction, and migrating the test base from one tool to another consumes engineering effort that could have been directed toward expanding coverage. The decision concerns the trajectory of the test base across the system's evolution, and a comparison that captures design and maintenance properties better supports this trajectory than one that records only the presence of features.

An additional source of difficulty is the gap between how a tool is demonstrated and how it is used. A short demonstration emphasizes the conciseness of a single request, while a production test base accumulates shared setup, data builders, and helper layers whose shape depends on the tool's model. Two tools that look equivalent in a one-line example diverge once auxiliary structures are added, and a comparison of working implementations exposes this divergence, whereas a comparison of isolated snippets conceals it. The present work follows this reading-based approach across three tools that share the same demonstration tasks.

The difficulty of the choice grows because many comparisons of API testing tools rely on vendor documentation or isolated examples that fail to reflect how a tool behaves as a test base grows [2]. The present work addresses this gap by building the comparison on working demonstration implementations that cover the same typical interface verification tasks.

The objective of the work is to qualitatively compare three approaches to automated API testing that differ in how an assertion is described and in how much of the protocol landscape they cover. The first approach under consideration is a fluent assertion syntax, represented by REST Assured, which combines an HTTP client with a domain-specific syntax for verifying Representational State Transfer (REST) interfaces. The second is a typed-client harness, represented by Retrofit, in which an annotated interface describes the service and the assertions are supplied by a separate library. The third is an action-based scenario engine, represented by Citrus, which supports several protocols and targets integration scenarios and message exchange. These three were selected because they span a meaningful range of the design space, from a lightweight assertion syntax through a typed client model to a multi-protocol action engine, and a comparison across this range illustrates the trade-offs that recur across the broader set of available tools.

The comparison is based on test design, readability, maintainability, and scope of application, and quantitative performance measurements are not conducted in the work, since the analysis focuses on design properties and applicability, not execution speed. The exclusion of performance figures is deliberate because such figures depend on the execution infrastructure, the size of the payload, and the configuration of the service under test, and they tell an engineer little about how a test reads or withstands a change to the interface contract. The criteria adopted here address the properties that persist across infrastructures and that govern the long-term cost of a test base.

The remainder of the text is organized as follows. The context of service-oriented architectures and the place of the interface verification level between the unit and integration levels are described first. The methodology of comparison and the set of criteria are stated next. The analysis of each tool and a summary comparison follow, and the text closes with a discussion of the applicability of the approaches and recommendations for choosing among them. Each analytical section returns to the same four criteria, which keeps the three tools commensurable and allows the summary comparison to rest on a uniform basis.

2. Background and related work

A service-oriented architecture organizes a system as a set of loosely coupled services with explicitly defined interfaces, and the microservice style extends this principle to a fine-grained division in which each service is deployed and scaled independently [3]. Different interaction styles coexist in such systems, and interface verification has to account for this heterogeneity.

The first widespread interaction style is REST over the Hypertext Transfer Protocol (HTTP), where resources are addressed using uniform methods, and the message body is serialized as JavaScript Object Notation (JSON) in most cases. Verification of this style focuses on the status code, the structure of the response body, and the headers that govern caching and content negotiation. The loose coupling between the body format and its schema means that a test often inspects individual fields by their path within the document. The second style is the exchange of Extensible Markup Language (XML) messages via the Simple Object Access Protocol (SOAP), which persists in enterprise and legacy systems. Verification of this style involves namespaces, envelopes, and schema definitions, and a tool that lacks native support for these constructs forces the engineer to assemble and parse the envelope by hand. The third style is asynchronous exchange via queues and message brokers, which complicates verification because responses arrive outside the original request. A test for this style has to express the dispatch of a message, the wait for a reply on a separate channel, and the correlation between the two. A tool whose model assumes a synchronous request and an immediate response fit this style awkwardly.

These three styles rarely occur in isolation. A system that began as a set of services exposing REST interfaces may acquire a gateway that speaks the simple object access protocol to an external partner, and a billing path that communicates through a message broker, and the test base then has to cover all three styles at once. An API testing tool is valuable to the extent that it covers the interaction styles present in a given system, and a tool confined to a single style obliges the team either to introduce a second tool or to extend the first with custom code. The breadth of protocol coverage is treated as a structural property rather than a secondary convenience in comparison.

The literature on test automation has long distinguished levels of verification, among which the unit level isolates individual classes through substitutes, and the integration level checks the joint operation of several services [4]. Interface verification occupies an intermediate position, since it accesses a service via its external boundary while requiring no startup of the entire system.

A separate line of research addresses the verification of interaction contracts, which reveals a mismatch between consumer and provider expectations before the integration stage, and this line adjoins the subject of interface verification, since both rest on a formal description of messages [5]. Studies of testing in microservice systems report that integration and inter-communication testing remain among the most active and challenging research themes, which places interface verification at the center of quality assurance for such systems [6]. The link between contract verification and interface verification matters for comparison, because a tool that clearly expresses the expected message also clearly expresses a contract, and the readability of an assertion is therefore more than a cosmetic property, since it bears on how early and how reliably a contract mismatch is caught.

The body of the listed works frames the way an API testing tool is assessed: namely, by which interaction styles it covers, at which verification level it operates, and how well its test description model aligns with maintenance tasks. The interface verification level deserves

particular attention within this frame, because it carries a property that neither the unit level nor the integration level carries on its own. It exercises the externally observable behavior of a service, remains faithful to the contract that other services depend on, and avoids the full system startup, so it runs at a speed and stability closer to those of unit tests. A tool that operates well at this level lets a team detect contract regressions early while keeping the feedback loop short, and the criteria that follow are chosen to surface how each tool behaves under exactly this dual demand. This frame determines the criteria of the subsequent comparison.

3. Methodology

The object of the analysis is three approaches to automated API testing, each represented by a reference implementation written for this study and reproduced in full below. The implementations run against a public reference service rather than a closed commercial system, which allows the approaches to be compared on identical material and lets any reader execute them, and this also sets the boundaries of the study. Each implementation solves the same small set of recurring tasks, among which are a verification of a successful read request, a verification of an error response, and a verification of a request that carries a body, so that the comparison rests on equivalent work and avoids tasks chosen to favor a particular approach.

The three subjects do not occupy identical roles in a project and presenting them as interchangeable products would misstate what is being compared. REST Assured is a testing library that combines an HTTP client with an assertion syntax. Retrofit is a typed HTTP client that carries no assertion facility of its own and becomes a testing approach only once an assertion library and a harness are built around it. Citrus is an integration testing engine that models a test as a sequence of message-exchange actions and brings its own runtime. The comparison is therefore drawn between three ways of expressing an interface test, and the unit of comparison is the test code that each way produces for one common scenario, not a feature-by-feature parity of the products. This framing also bounds the claims made below, since a statement about an approach describes the shape of the resulting test and does not assert that one product subsumes another.

The method of comparison rests on reading working implementations in which the same typical interface verification tasks are solved by each tool, thereby distinguishing the approach from a comparison based on capability descriptions in vendor documentation [7]. The analysis is qualitative, yielding a structured comparison of properties.

3.1. Common test scenario and reference implementations

A single scenario is used across the three approaches so that every observation rests on equivalent work. The scenario is defined against the Swagger Petstore service, a publicly hosted reference implementation of an OpenAPI-described REST interface, which removes any dependence on closed systems, supplies a machine-readable contract of the kind a typed client consumes, and lets a reader run the implementations without additional setup.

The scenario consists of three cases that occur in interface verification. Case S1 reads an existing resource: a request for a pet by its identifier is expected to return to success status and a body whose identifier matches the request and whose name is not empty. Case S2 exercises an error path: a request for an identifier that does not exist is expected to return a not-found status together with an error body carrying the corresponding message. Case S3 sends a payload: a request that creates a pet is expected to return a success status and a

body that echoes the submitted name and status. The three cases cover a successful read, an error response, and a request that carries a body, and they are held fixed across the three approaches, so that no case is chosen to favor a particular one.

The reference implementations are reproduced in full in Section 4 and are pinned to the versions used here: REST Assured 5.5.0, Retrofit 2.11.0 with a Jackson converter and JUnit assertions, and Citrus 4.5.0, all on JUnit 5 and Java 17. Each implementation solves cases S1 to S3 and nothing beyond them, so that differences in length and structure between them follow from the approach rather than from differences in scope. The listings omit package declarations and import statements for space, and the three declarations of Listing 2, which belong to separate compilation units, are shown together.

3.2. Criteria and observable indicators

The set of comparison criteria includes several dimensions. The first dimension is test design, understood as the way a request, an expected response, and the assertions are described. Test design covers the granularity at which the engineer works: one tool exposes the raw request and response, while another wraps them in typed abstractions, and it covers the locus of validation: one tool checks the contract at runtime, while another shifts part of the check to the compilation stage. The second dimension is readability, which reflects how clear a test description is to an engineer who did not write it. Readability depends on whether the test reads in the order of the interaction it describes whether the intent of an assertion is visible without tracing helper code, and the volume of incidental detail the tool's model requires.

The third dimension is maintainability, which covers the resilience of tests to interface change and the reuse of shared parts. Maintainability is observed through the number of places a single contract change touches, whether a mismatch surfaces as a compilation failure or a runtime assertion, and the ease with which common setup and message templates are factored out of individual tests. The fourth dimension is the scope of application, which describes which interaction styles and which verification levels a tool covers. Scope of application records native support for REST, the simple object access protocol, and asynchronous messaging, and whether the tool addresses a single service in isolation or orchestrates exchanges across several endpoints. The four dimensions are not independent, since a broader scope often leads to verbosity that lowers readability, and the discussion returns to these tensions when the tools are compared.

Each criterion is tied to an indicator that can be read directly from an implementation, so that a qualitative judgement is anchored in something a reader can check against the listings rather than in an impression. Table 1 states the four criteria, the indicator adopted for each, and the evidence in the listings from which that indicator is read.

Table 1: Comparison criteria and the indicators used to read them from an implementation

Criterion	Observable indicator	Evidence read from the implementation
Test design	Level at which the exchange is manipulated, and the stage at which the contract is declared	Whether the listing names raw HTTP elements, typed methods, or explicit actions, and where the endpoint and the payload shape are written
Readability	Number of separate declarations a reader must open to reconstruct one case	Whether case S1 can be understood from the test method alone, or requires the interface, the model, or the endpoint configuration

Criterion	Observable indicator	Evidence read from the implementation
Maintainability	Number of sites a single contract change touches, and the stage at which a mismatch surfaces	Count of assertion sites bound to the response shape, and whether renaming a field breaks compilation or only the run
Scope of application	Interaction styles expressed natively, and whether one or several endpoints are orchestrated	Presence of native REST, SOAP, and asynchronous messaging in one model, and of a construct spanning several exchanges

Measurements of performance and throughput are not conducted in the work, since they depend on the execution infrastructure and do not characterize the design approaches themselves. Any quantitative coverage indicators and other metrics of closed projects are also excluded from the analysis, which aligns with the review and analytical character of the work. The reading of working implementations replaces the collection of numerical data, and the unit of evidence is a passage of test code that solves one of the recurring tasks, examined for what it reveals about design, readability, maintainability, and scope. This approach has a known weakness: a reading reflects the reader's judgment, and the work mitigates this by holding the four criteria fixed across all three tools and by drawing every observation from the same set of tasks. The role of the interface verification level in the overall testing structure of a service-oriented system is shown in Figure 1 below.

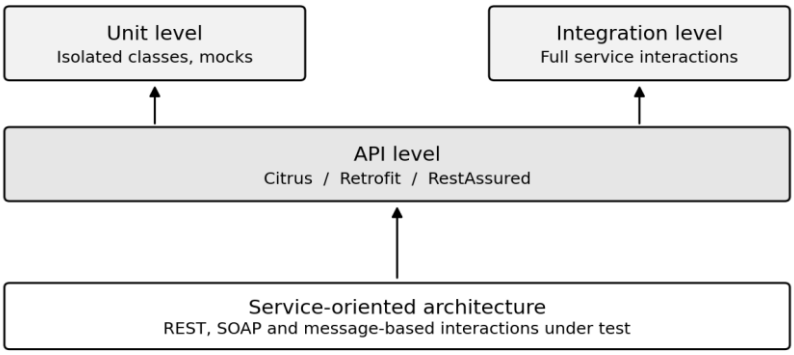

Figure 1: Positioning of API-level testing between unit and integration levels in a service-oriented architecture

Figure 1 shows that the interface verification level is based on the interaction styles of the service-oriented system and the way it accesses services through its external boundary, while it falls between the isolated unit level and the full integration check [4]. The arrows in the figure point upward from the architecture to the verification levels, indicating that the interaction styles present in the system determine what tests at each level can observe. A tool placed at the interface level inherits this dependence, so its value is bounded by how well its model matches the styles that the architecture exhibits, and the three tools studied here differ precisely in the breadth of that match.

4. Results

The analysis of each approach rests on the same set of criteria and on the reference

implementations of the common scenario described in Section 3.1. Each subsection first shows how the approach expresses cases S1 to S3, then reads the indicators of Table 1 off that code, and then evaluates the approach against test design, readability, maintainability, and scope of application. The order of the subsections follows the design space from the lightest assertion syntax to the broadest action engine, so that the reader meets the trade-off between conciseness and breadth as a gradual progression across the three approaches.

4.1. REST Assured

REST Assured describes the verification of a REST interface through a chain of calls, laid out as a domain-specific syntax that reads as a sequence of request preparation, request dispatch, and response checking [8]. Request preparation sets the base address, the headers, and the parameters; dispatch is expressed via a separate link in the chain; and assertions are formulated as statements about the status and the body of the response.

Listing 1: Cases S1 to S3 expressed with REST Assured, a fluent assertion syntax

```
class PetApiRestAssuredTest {

  @BeforeAll
  static void setUp() {
    RestAssured.baseURI = "https://petstore.swagger.io/v2";
  }

  @Test // S1: read an existing resource
  void readsExistingPet() {
    given().pathParam("id", 1)
    .when().get("/pet/{id}")
    .then().statusCode(200)
          .body("id", equalTo(1))
          .body("name", not(emptyString()));
  }

  @Test // S2: error path
  void reportsMissingPet() {
    given().pathParam("id", 9999999)
    .when().get("/pet/{id}")
    .then().statusCode(404)
          .body("message", equalTo("Pet not found"));
  }

  @Test // S3: request carrying a body
  void createsPet() {
    given().contentType(ContentType.JSON)
          .body(Map.of("id", 77, "name", "Rex", "status", "available"))
    .when().post("/pet")
    .then().statusCode(200)
          .body("name", equalTo("Rex"))
          .body("status", equalTo("available"));
  }
}
```

Read against Table 1, the listing places the engineer at the level of the raw exchange: the path, the query parameters, and the status code appear literally, and every expectation about the body is a string naming a position inside the JSON document. One file and one method hold a complete case, so a reader reconstructs case S1 without opening any other declaration. The same property fixes the maintainability indicator, since the six path strings across the three cases each restate a part of the contract, and a renamed field is detected only when the test runs.

The readability of such a description is high for REST interfaces, since the test structure mirrors the natural flow of a request to the service, and the engineer can see the verified behavior without reference to auxiliary code. Maintainability remains acceptable, though

assertions in the response body bind to paths within the data structure in many cases, and a change to the response format requires a coordinated edit of these paths. The tool's scope of application focuses on synchronous REST interfaces, while message exchange in Extensible Markup Language and asynchronous scenarios fall outside its core model.

Against the four criteria, the tool shows a consistent profile. In test design, it operates at the level of the raw request and response, giving the engineer direct control over headers, query parameters, and the inspection of arbitrary fields. This directness suits services whose responses vary in shape from one endpoint to the next. In terms of readability, it benefits from alignment between the chain of calls and the sequence of preparation, dispatch, and assertion, so a reader can follow the test as a narrative of a single request. In terms of maintainability, it carries a cost that grows with the depth of the response structure, because each path expression is a point at which a format change can break, and a demonstration with shallow responses understates this cost relative to a service with deeply nested documents. In scope, it remains a specialist of synchronous REST, and a team that adopts it for a REST-only system gains conciseness, while a team facing mixed protocols inherits the task of covering the remaining styles by other means.

4.2. Retrofit

Retrofit is a typed client that uses a set of annotated methods to describe the service interface and translates calls into requests to the remote interface. In the test harness, this client is extended with auxiliary layers that hide the repeated details of a request [9]. The description of a test under such a design relies on typed methods, which move some checks to the compilation stage.

Listing 2: Cases S1 to S3 expressed with Retrofit, a typed-client harness

```
interface PetApi {

  @GET("pet/{id}")
  Call<Pet> getPet(@Path("id") long id);

  @POST("pet")
  Call<Pet> addPet(@Body Pet pet);
}

record Pet(long id, String name, String status) {}

class PetApiRetrofitTest {

  private static PetApi api;

  @BeforeAll
  static void setUp() {
    api = new Retrofit.Builder()
        .baseUrl("https://petstore.swagger.io/v2/")
        .addConverterFactory(JacksonConverterFactory.create())
        .build()
        .create(PetApi.class);
  }

  @Test // S1: read an existing resource
  void readsExistingPet() throws IOException {
    Response<Pet> response = api.getPet(1).execute();
    assertEquals(200, response.code());
    assertEquals(1, response.body().id());
    assertNotNull(response.body().name());
  }

  @Test // S2: error path
  void reportsMissingPet() throws IOException {
    Response<Pet> response = api.getPet(9999999).execute();
```

```
    assertEquals(404, response.code());
    assertNull(response.body());
  }

  @Test // S3: request carrying a body
  void createsPet() throws IOException {
    Response<Pet> response =
        api.addPet(new Pet(77, "Rex", "available")).execute();
    assertEquals(200, response.code());
    assertEquals("Rex", response.body().name());
    assertEquals("available", response.body().status());
  }
}
```

Read against Table 1, the listing raises the level of abstraction above the raw exchange: the endpoint, the method, and the shape of the payload are declared once in the annotated interface and the record, and each case calls a named method and asserts on typed accessors. The cost appears in the readability indicator, since reconstructing case S1 requires three declarations rather than one.

The maintainability indicator distinguishes this approach from the other two, though with a qualification that matters. A contract change is written in one place, and once the record is updated the compiler flags every call site that no longer matches, which converts a single edit into a complete list of the code it affects. The compiler does not, however, observe the remote service: if the provider renames a field and the local declaration is left untouched, the code still compiles and the mismatch surfaces at run time, exactly as in the other two approaches. The advantage is therefore the concentration of the contract in a single declaration and the propagation of an edit from it, and not an early warning about the service itself.

Readability is supported by the fact that a request to the service is expressed by a call to a named method, and the request and response structures are fixed by types, reducing the need for explicit checks of individual fields. Maintainability is reinforced by a single description of the interface, since a change to the contract is reflected in a single place, and the dependent code stops compiling, which exposes the mismatch early. The scope of application covers synchronous REST interfaces, and support for message exchange in the extensible markup language falls outside the principal model of the tool; such scenarios are addressed beyond it through harness means when the need arises.

Against the four criteria, the tool trades directness for structure. In test design, it raises the level of abstraction above the raw request, since the typed interface describes the endpoint once, and the test calls a method, and this abstraction suits a service whose contract is stable enough to be worth encoding in types. In readability, it benefits from the absence of repeated low-level detail, though a reader must consult the interface declaration to learn the exact shape of the request, so understanding is split between the test and the interface. In maintainability, it offers the strongest position of the three for a changing REST contract, because the compiler flags every call that no longer matches the interface, and mismatches surface before a test runs. In scope, it shares the REST specialization of the first tool, so the same boundary applies, and the asynchronous and the simple object access protocol styles call for separate provision.

4.3. Citrus

Citrus describes a test as a sequence of actions, among which the dispatch of a message and the receipt of a response are expressed by separate steps, and this model extends to several protocols, including REST, message exchange in the extensible markup language, and asynchronous queues [10]. The actions are combined into a scenario that reproduces the order of exchange between services.

Listing 3: Cases S1 to S3 expressed with Citrus, an action-based scenario engine

```
@CitrusSupport
class PetApiCitrusTest {

  private final HttpClient petstore = http().client()
      .requestUrl("https://petstore.swagger.io/v2")
      .build();

  @Test // S1: read an existing resource
  @CitrusTest
  void readsExistingPet(@CitrusResource TestCaseRunner t) {
    t.when(http().client(petstore).send().get("/pet/1"));
    t.then(http().client(petstore).receive().response(HttpStatus.OK)
        .message().type(MessageType.JSON)
        .expression("$.id", "1")
        .expression("$.name", "@notEmpty()@"));
  }

  @Test // S2: error path
  @CitrusTest
  void reportsMissingPet(@CitrusResource TestCaseRunner t) {
    t.when(http().client(petstore).send().get("/pet/9999999"));
    t.then(http().client(petstore).receive().response(HttpStatus.NOT_FOUND)
        .message().type(MessageType.JSON)
        .expression("$.message", "Pet not found"));
  }

  @Test // S3: request carrying a body
  @CitrusTest
  void createsPet(@CitrusResource TestCaseRunner t) {
    t.when(http().client(petstore).send().post("/pet")
        .message().contentType("application/json")
        .body("{\"id\":77,\"name\":\"Rex\",\"status\":\"available\"}"));
    t.then(http().client(petstore).receive().response(HttpStatus.OK)
        .message().type(MessageType.JSON)
        .expression("$.name", "Rex")
        .expression("$.status", "available"));
  }
}
```

Read against Table 1, the listing works at the level of explicit actions: the dispatch of the request and the receipt of the response are separate statements, and nothing in the model requires them to be adjacent. For a single synchronous request this separation is pure cost, and the three cases occupy more lines than the fluent syntax needs for the same work. The separation earns its keep where a reply arrives on a different channel from the request, or where several exchanges have to be ordered, because the same construct then expresses what a single chained call cannot. On the maintainability indicator the approach sits with the first, since five path expressions restate parts of the contract and a mismatch appears when the scenario runs.

The description's readability decreases due to verbosity, as each step of the exchange is expressed explicitly. This same explicitness makes the scenario clear when a complex interaction is examined, where the order of messages matters. Maintainability is supported by the reuse of actions and message templates, which allows long scenarios to be assembled from shared parts. The scope of application covers integration scenarios and message-based interaction, and the multi-protocol property is what distinguishes the tool from those focused on REST interfaces.

According to the four criteria, the tool reverses the profile of the first two. In test design, it works at the level of explicit actions, so a test names the dispatch of a message and the receipt of a reply as separate steps, and this granularity matches an interaction whose meaning lies in the order and the channels of the exchange. In readability, it pays a verbosity cost for a

single REST request, where the explicit steps add length that a fluent chain avoids, yet it recovers this cost for a multi-step exchange, where the same explicitness records a sequence that a fluent chain would obscure. In maintainability, it relies on the factoring of repeated actions and message templates, and a scenario assembled from shared parts withstands change in proportion to how well these parts isolate the points that change. In scope, it stands apart as the only one of the three with native coverage of REST, the simple object access protocol, and asynchronous messaging, and this breadth is what recommends it for a heterogeneous system. The difference between description via a domain-specific syntax and via a sequence of actions is shown in Figure 2 below.

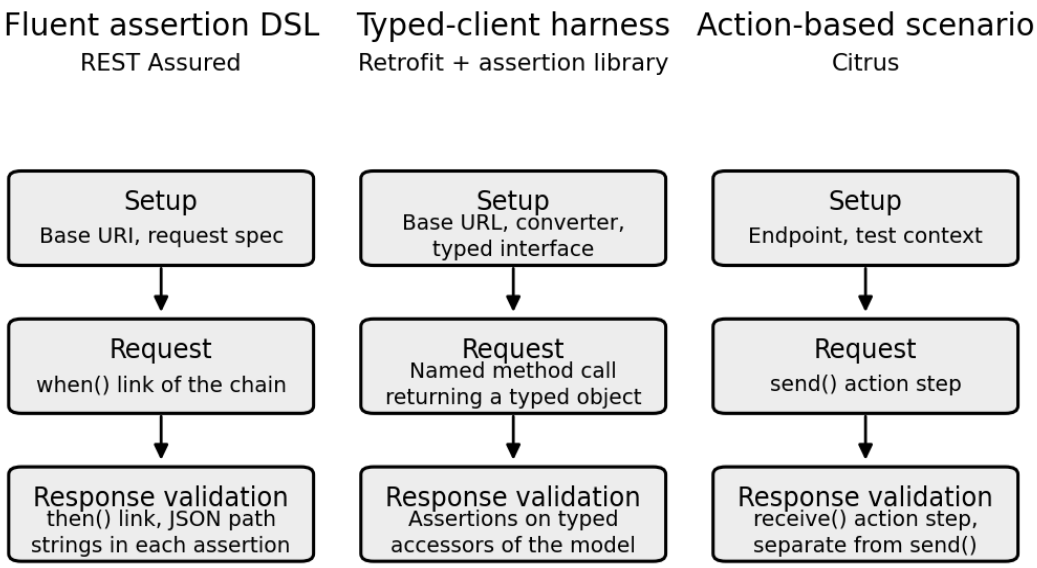

Figure 2: Structural comparison of a single API test across the three approaches: fluent assertion syntax, typed-client harness, and action-based scenario

Figure 2 shows that all three approaches pass through the same phases of preparation, request dispatch, and response validation, and that they differ in how these phases are bound together and in where the contract is written down. The fluent syntax binds the request and the validation into a single chain and restates the contract in every assertion. The typed-client harness separates a named request from assertions on typed accessors and declares the contract once, in the interface and the model. The action-based scenario makes dispatch and receipt independent steps and restates the contract in every message expression. The consequence of this structural difference reaches beyond a single test. When the phases are bound into a chain, the test reads as a short narrative, and the dispatch and the check remain adjacent, which favors a single synchronous request. When they are separate steps, additional steps can be inserted between them, which favors an exchange in which a message travels on one channel and a reply returns on another. The same separation that lengthens a simple REST test is what makes an asynchronous or multi-message scenario expressible, and the figure therefore captures a trade-off rather than a ranking.

4.4. Indicators read from the implementations

Because the three listings solve one scenario, the indicators of Table 1 can be compared directly. Table 2 collects them. The counts refer to the implementations of Listings 1 to 3 with package declarations and import statements excluded, and they describe three cases only, so

they should be read as a relative ordering rather than as a projection onto a production test base.

Table 2: Indicators of Table 1 as read from the reference implementations of cases S1 to S3

Indicator	REST Assured	Retrofit	Citrus
Lines of code for cases S1 to S3	30	39	34
Files required	1	3	1
Declarations to open to read case S1	1	3	1
Sites bound to the response shape	6 JSON path strings	5 typed accessors	5 JSONPath expressions
Where the contract is declared	In every assertion	Once, in the interface and the model	In every message expression
Effect of renaming a response field	Run-time assertion failure	Compilation failure at each call site once the model is updated	Run-time validation failure

The counts place the fluent syntax first on volume and on the number of declarations a reader opens, the action-based engine close behind on volume but with the request and the response held apart, and the typed-client harness last on both counts, because the contract is lifted into declarations of its own. That ordering inverts on the last row, where the same lifting is what allows one edit to be propagated by the compiler across the test base. No approach leads on every indicator, and the table therefore supports a choice conditioned on the system rather than a ranking.

5. Discussion

The comparison of the three approaches using the adopted criteria shows that they occupy different niches within the interface verification level, with differences driven by protocol coverage and the test description model. The summary comparison is presented in Table 3 below.

Table 3: Qualitative comparison of the three API-testing approaches across design and applicability criteria

Criterion	REST Assured	Retrofit	Citrus
Protocol focus	REST over HTTP	REST over HTTP	REST, SOAP, messaging
Test definition	Fluent domain-specific language (DSL) chain	Typed interface and helpers	Action-based scenario
Readability	High for REST	High via typed methods	Verbose but explicit
Maintainability	Coupled to data paths	Single interface source	Reusable action templates
SOAP support	Limited	Not native	Native
Integration scope	Single service	Single service	Multi-endpoint orchestration
Best fit	Lightweight REST checks	Typed client-driven tests	Mixed-protocol integration

Table 3 shows how protocol coverage sets the main line of distinction between the approaches. REST Assured and Retrofit focus on synchronous REST interfaces, and their strength lies in concise test descriptions, whereas Citrus supports multiple protocols and therefore suits scenarios where REST, message exchange in the Extensible Markup

Language, and asynchronous queues coexist. A second line of difference runs through the locus of validation, since the typed-client harness concentrates the contract in a declaration from which the compiler propagates an edit, while the other two restate it at each assertion and detect a mismatch when the test runs. A third line runs through the granularity of description, from the fluent chain of the first approach through the typed method of the second to the explicit action of the third.

Reading the table across rows, and not only down columns, reveals how the criteria interact. The protocol focus row and the integration scope row move together, since the tool with the widest protocol coverage is also the one that orchestrates several endpoints, and this coupling reflects a single underlying property: whether the tool's model assumes a single synchronous request or a sequence of exchanges. The readability row and the test definition row move in tension, since the typed and the fluent models gain conciseness for a single REST request while the action model gains clarity for a multi-step exchange, and no single entry in the readability row holds across every interaction style. The maintainability row distinguishes the early compile-time detection of the typed model from the runtime detection of the other two, and this distinction matters most where the contract changes often.

The applicability of each approach depends on the type of interaction in a given system. When a service exposes a REST interface and is verified in isolation, a concise description in a domain-specific syntax reduces code volume and speeds up failure analysis. When the interface contract changes frequently, and early detection of mismatches matters, the typed client moves some checks to the compilation stage, reducing the risk of unnoticed divergences. When the interaction of several services across different protocols is verified, a description in terms of a sequence of actions conveys the exchange order with greater precision, and the multi-protocol property becomes a decisive advantage.

Several concrete situations illustrate how these considerations combine. A team building a small set of REST services with stable contracts gains the most from a fluent syntax, since the conciseness shortens both the writing and the reading of tests, and the stability of the contracts keeps the cost of runtime path assertions low. A team whose REST contracts evolve under active development gains from the typed client, since the compiler converts each contract change into a visible signal and reduces the number of failures discovered only at test execution. A team integrating services across REST, the simple object access protocol, and a message broker gains from the action-based engine, since a single tool then covers the whole landscape, and the explicit steps document the order of a multi-protocol exchange. A team whose landscape combines these conditions is justified in combining tools, applying a fluent or typed syntax to isolated REST checks, and using an action-based engine for integration scenarios that cross protocols.

The combination of tools carries its own cost, since two tools mean two sets of conventions, two harness layers, and two bodies of knowledge that the team keeps current, and this cost weighs against the awkwardness of forcing a single tool onto a style it was not built for. The boundary between a single-tool decision and a multi-tool decision depends on how large and how lasting the off-style portion of the system is, so a small or transient portion argues for extending one tool, while a large and lasting portion argues for adopting a second. This reasoning treats tool choice as a property of the system landscape and reaches a recommendation by matching the profiles in the summary table to the styles in the landscape.

The comparison has limitations that should be taken into account when drawing conclusions. The analysis rests on reference implementations of a three-case scenario, and the behavior of the approaches in a large production test base may differ in the volume of

auxiliary harness and in the cost of maintenance, so the counts in Table 2 order the approaches rather than predict their cost at scale. The three subjects also occupy different roles, since one is a testing library, one an HTTP client wrapped in a harness, and one an integration engine, so a statement about an approach describes the shape of the test it produces and is not a verdict on a product. All three are drawn from the Java ecosystem, and the observations are therefore not carried over to tools of other language ecosystems without further verification. The qualitative nature of the criteria means that a preference between approaches that are close in their properties depends on team practices and the established technology base. A further limitation is that the comparison fixes a single version of each tool and a single style of use, while each admits configurations and extensions that shift its profile, so the entries in the summary table describe a representative use and not the full envelope of what a tool can be made to do. These limitations outline the applicability of the results and provide a direction for further verification on larger materials.

6. Conclusion

The comparison of three API-testing approaches shows that the choice in a service-oriented environment depends on the type of interaction and the maintenance horizon of the test base, rather than on a universal superiority of one approach over the others. A flexible domain-specific syntax supports concise verification of REST interfaces; a typed client concentrates the contract in a single declaration from which the compiler propagates an edit across the test base; and an engine supporting several protocols covers integration scenarios and message exchange where different interaction styles coexist.

The practical meaning of these observations is that an engineer selects a tool based on the interaction styles of the verified system and the expected lifetime of the tests, and that combining several tools across different segments is warranted when the interaction landscape is heterogeneous. The summary comparison offers a starting point for this selection, since it maps each tool to the conditions under which its profile is strongest, and a team can read the table against the properties of its own system to narrow the choice before any code is written.

The contribution of the work is a direct comparison of reference implementations of one common scenario within a single analytical frame, evaluated against criteria tied to indicators observable in the code, which replaces a feature-by-feature reading of vendor documentation with a reading of how the approaches behave on equivalent tasks. Further work involves the verification of the stated observations on a larger test base, the extension of the set of criteria with dimensions tied to maintenance cost under long operation, and a study of how the profiles shift when the tools are configured and extended beyond their representative use.

Declarations

Ethics approval: Not applicable.

Funding: No external funding.

Competing interests: The author declares no competing interests.

Author contributions: The author contributed to the study and approved the final manuscript.

References

[1] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges," IEEE Access, vol. 11, pp. 88339–88358, 2023, doi: 10.1109/ACCESS.2023.3305687.

[2] A. Golmohammadi, M. Zhang, and A. Arcuri, "Testing RESTful APIs: A survey," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 1, Nov. 2023, Art. no. 27, doi: 10.1145/3617175.

[3] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[4] M. Cohn, Succeeding with Agile: Software Development Using Scrum, 1st ed. Boston, MA, USA: Addison-Wesley Professional, 2009.

[5] J. Lehvä, N. Mäkitalo, and T. Mikkonen, "Consumer-driven contract tests for microservices: A case study," in Proc. 20th Int. Conf. Product-Focused Softw. Process Improvement (PROFES), Barcelona, Spain, 2019, pp. 497–512, doi: 10.1007/978-3-030-35333-9_35.

[6] M. Waseem, P. Liang, G. Márquez, and A. Di Salle, "Testing microservices architecture-based applications: A systematic mapping study," in Proc. 27th Asia-Pacific Softw. Eng. Conf. (APSEC), Singapore, 2020, pp. 119–128, doi: 10.1109/APSEC51365.2020.00020.

[7] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated test generation for REST APIs: No time to rest yet," in Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA), Virtual, South Korea, 2022, pp. 289–301, doi: 10.1145/3533767.3534401.

[8] J. Haleby. "REST Assured documentation and usage guide." REST Assured project. Accessed: May 25, 2026. [Online]. Available: https://rest-assured.io/

[9] "Retrofit: A type-safe HTTP client for Android and the JVM." Square, Inc. Accessed: May 31, 2026. [Online]. Available: https://square.github.io/retrofit/

[10] "Citrus Framework reference documentation: Automated integration testing for message-based applications and microservices." Citrus Framework, release 4.x. Accessed: Jun. 7, 2026. [Online]. Available: https://citrusframework.org/