International Journal of Advanced Research in Computer Science & Technology (IJARCST)

| ISSN: 2347-8446 | www.ijarcst.org | editor@ijarcst.org |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January-February 2023||

DOI:10.15662/IJARCST.2023.0601003

Structuring Reusable API Testing Frameworks with Cucumber-BDD and REST Assured

Chiranjeevulu Reddy Kasaram

Independent Researcher, USA

chiran.reddy16@gmail.com

ABSTRACT*:* API testing is central to ensuring the reliability and stability of modern service-driven applications, yet many automation solutions become brittle and difficult to maintain as systems evolve. This paper presents an architectural approach for building reusable and scalable API testing frameworks by integrating Cucumber-BDD with REST Assured. Leveraging Cucumber's Gherkin language for business-readable specifications and REST Assured's fluent DSL for HTTP request validation, the proposed layered architecture separates feature definitions, step implementations, and reusable service components. This design enables reusability, minimizes code duplication, and supports maintainability by isolating changes to specific modules. By aligning business requirements with technical execution, the framework enhances readability, accelerates onboarding, and ensures portability across environments and CI/CD pipelines. The result is a sustainable approach to API test automation that balances expressiveness, technical rigor, and long-term maintainability

KEYWORDS: ***API testing, Cucumber-BDD, REST Assured, test automation, scalability***

I. INTRODUCTION

The combination of Cucumber-BDD and REST Assured would require a properly organized structure to create reusable and maintainable API test automation. The architecture uses the Gherkin language of Cucumber to establish executable specifications, which can be read and understood by humans, to bridge the communication gap between the technical and non-technical stakeholders [3]. These behavioural scenarios are then converted into specific actions via the REST Assured potent, fluent DSL of making HTTP requests and making assertions [1]. A layered architecture is essential to achieve maximum reusability with the separation of the test specification (feature files), glue code(step definition), and reusable service layer component which defines API interactions. With strong models of request/response payloads, this design guarantees that modifications of the API contract need to make the least changes in the codebase, which facilitates scalability and simple maintenance in an agile/cloud-based setup [1], [4].

II. BACKGROUND CONCEPTS: CUCUMBER-BDD AND REST ASSURED

The suggested framework is effective due to the two basic technologies being combined synergistically, specifically, Cucumber to Behaviour-Driven Development (BDD) and REST Assured to the API interaction. BDD is an integrated practice that builds on Test-Driven Development (TDD) by writing tests that are expressed in a common language that can be understood by all project stakeholders, therefore taking development efforts into consideration of the intended business results [4]. Cucumber implements BDD by supporting the Gherkin language a plain-text based, simple and structured syntax of Given, When, and Then steps specifying application behaviour in plain-text feature files. This takes test cases and turns them into executable specifications which can be used both as documentation and as verification suites [5].

Complementing this, REST Assured is a Java DSL designed specifically for simplifying the testing of REST-based services. It provides a highly expressive, fluent interface that allows testers to craft complex HTTP requests—including setting headers, parameters, and body content—and to validate responses with powerful assertions, all in a manner that reads like a natural language sentence [1]. Its deep integration with Groovy enables elegant parsing and navigation of JSON and XML response structures, eliminating the need for verbose boilerplate code.

The true power emerges when these tools are integrated. Cucumber's Gherkin scenarios define the what—the expected behaviour of the API from a functional perspective. REST Assured, invoked within the step definitions, implements the how—the technical execution of the HTTP calls and validations. This clear separation of concerns is the bedrock upon

International Journal of Advanced Research in Computer Science & Technology (IJARCST)

| ISSN: 2347-8446 | www.ijarcst.org | editor@ijarcst.org |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January-February 2023||

DOI:10.15662/IJARCST.2023.0601003

which a reusable and maintainable framework is built, ensuring that business logic remains distinct from the underlying technical implementation [7].

III. THE PILLARS OF A REUSABLE FRAMEWORK ARCHITECTURE

Before delving into implementation, it is crucial to define the core architectural principles that guide the framework's design. A reusable API testing framework must be constructed with specific, non-negotiable pillars in mind to avoid the common pitfalls of brittleness and high maintenance. These pillars ensure the framework's longevity and Return on Investment (ROI) as the application under test evolves.

The primary goal is Reusability, which dictates that code components should be written once and utilized across multiple test scenarios. This is achieved by abstracting common operations, such as authentication or request specification building, into shared modules, preventing costly code duplication [1]. Directly tied to this is Maintainability. A well-architected framework ensures that a change in the API contract—such as a new mandatory header or a modified endpoint—requires a change in only one location within the codebase, significantly reducing effort and risk [2].

Furthermore, the framework must champion Readability and Clarity. The use of Cucumber-BDD inherently addresses this at the specification level, but the underlying code must also be clean and intuitive. This allows new team members to onboard quickly and makes test debugging a more straightforward process [5]. Finally, the architecture must be Scalable and Portable. It must support a growing number of tests and APIs without a degradation in performance or organization, and it must be easily executable across different environments (e.g., development, staging, production) and within CI/CD pipelines without modification [1], [4].

TABLE I. COMPARISON OF DATA REQUIREMENTS: TRADITIONAL VS. AGENTIC AI SYSTEMS

Pillar	Description	Key Benefit
Reusability	Designing components (e.g., request builders, utility functions) for broad consumption across tests.	Reduces code duplication, accelerates test script development.
Maintainability	Isolating volatile code (e.g., endpoints, headers) into centralized, single-responsibility classes.	Minimizes the impact of API changes, lowering long-term maintenance costs.
Readability	Combining Gherkin's business language with well-named, logical code structures.	Enhances collaboration and simplifies debugging and review processes.
Scalability	Structuring the framework in discrete layers to manage complexity and growth.	Supports an expanding test suite efficiently without structural overhaul.

Deep Dive: Critical Components and Technologies

International Journal of Advanced Research in Computer Science & Technology (IJARCST)

| ISSN: 2347-8446 | www.ijarcst.org | editor@ijarcst.org |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January-February 2023||

DOI:10.15662/IJARCST.2023.0601003

Translating the theoretical pillars of reusability, maintainability, and scalability into a practical, implementable system requires a deliberate and layered architectural approach. This structure is paramount, as it enforces a clean separation of concerns, ensuring that each component within the framework has a single, well-defined responsibility. The proposed blueprint, visualized in Figure 1, consists of four distinct layers—each building upon the one below it—and a supporting infrastructure that enables cross-cutting functionality. This hierarchy is designed to isolate volatility, promote code reuse, and create a sustainable ecosystem for test development that can evolve alongside the application it validates [1], [5].

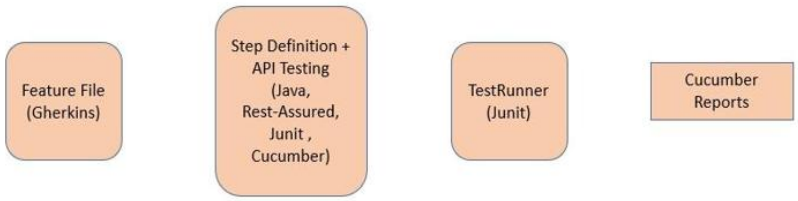

Figure 1. The layered architecture of a Cucumber-REST Assured testing framework, depicting the flow from business-readable features to executable code

Layer 1: The Core Framework Layer

This foundational layer houses all low-level, reusable components that are entirely agnostic to specific API endpoints or business logic. Its stability is critical, as changes here can ripple throughout the entire framework.

- Request Specification Builder: This is perhaps the most crucial component for ensuring maintainability. A central class (e.g., RequestSpecBuilder) is responsible for constructing and pre-configuring all HTTP requests. It encapsulates volatile and common configurations such as the base URI, default headers (e.g., Content-Type: application/json), authentication mechanisms (e.g., adding Oauth2 tokens or API keys), proxy settings, and logging preferences. By serving as the single source of truth for request configuration, a modification like a new mandatory authentication header requires a change in only this one location, instantly propagating the update to every test in the suite [1]. This design directly combats test brittleness.
- Data Model Classes (POJOs/Java Records): These classes provide a type-safe, object-oriented representation of the JSON or XML request payloads and response objects exchanged with the API. Using Plain Old Java Objects (POJOs) annotated with Jackson or modern Java Records, testers can model API entities like User, Product, or Order. This approach eliminates the brittleness and verbosity of manually constructing JSON strings and parsing responses using complex, string-based JSONPath queries. A library like Jackson handles the serialization (Java object to JSON) and deserialization (JSON to Java object) seamlessly, leading to cleaner, more reliable, and more readable code [5]. For instance, asserting response.getUser().getFirstName() is far more intuitive than response.jsonPath().getString("user.firstName").
- Utility Classes: A suite of helper classes provides common, cross-cutting functionalities. This includes a ConfigReader to parse environment-specific properties from files (e.g., dev.properties, prod.properties), a TestDataGenerator using libraries like Java Faker to create dynamic and random test data, and custom assertion classes that extend REST Assured's validation capabilities for complex, multi-field response validations.

TABLE II. RAG PROCESS FOR A FINANCIAL ANALYST AGENT

Component	Primary Responsibility	Example
RequestSpecBuilder	Pre-configures all HTTP requests with base settings (URI, auth, headers).	RequestSpecification requestSpec = new RequestSpecBuilder().setBaseUri(config.getBaseUrl()).addHeader ("Authorization", "Bearer " + token).build();
User.java (Record)	Models the structure of a User entity for request/response payloads.	public record User(int id, String name, String email, String password) {}

International Journal of Advanced Research in Computer Science & Technology (IJARCST)

| ISSN: 2347-8446 | www.ijarcst.org | editor@ijarcst.org |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January-February 2023||

DOI:10.15662/IJARCST.2023.0601003

ConfigReader.java	Reads parameters (e.g., URLs, credentials) from external .properties or .yaml files.	String baseUrl = ConfigReader.getProperty("api.base.url");
TestDataGenerator .java	Generates dynamic, random data for test inputs to ensure uniqueness.	String randomEmail = TestDataGenerator.generateRandomEmail(); // e.g., "user_123@test.com"

Layer 2: The Helper/Service Layer

The layer serves as an inseparable point between the generic basic utilities and the particular stages of the tests. It wraps all the communication with certain API endpoints into service client classes that can be reused without considering the technicalities of HTTP in the application of the test logic itself [2], [7].

- Purpose and Implementation: API resources (e.g. User, Product, Order) are assigned to services which represent their classes (e.g. UserAPIService). These classes provide advanced techniques to all the operations which can be performed on that resource, i.e, createUser(User user), getUserById(int id), and updateUser(int id, User user). Within these techniques, the REST Assured code is coded, using the ready-to-use RequestSpecification of Layer 1 and returning the API Response or a deserialized POJO.

Advantages:

This abstract is strong. It implies that the step definitions in Layer 3 are made extremely thin, declarative, and behavior-oriented, instead of implementation-oriented. These service methods are merely called with ready data. This layer includes all the complex code of the REST Assured - the construction of the endpoint URL, selecting the HTTP method, and body, and the implementation of the request. This allows the service techniques to be extremely reusable in a variety of test cases and protect the test cases against modifications in the API HTTP interface.

Layer 3: The Step Definitions Layer

This layer contains the "glue" code that maps the natural language Gherkin steps from the feature files to executable Java code. Each Given, When, and Then step has a corresponding method that implements its intent.

- Implementation: Step definition methods should be concise. Their primary role is to extract parameters from the Gherkin step, potentially transform them into POJOs, call the appropriate method in the Service Layer (Layer 2), and store responses or perform assertions on return values. They translate the "what" into "how" by delegating the actual work to the layers below.
- Data Handling: This layer efficiently leverages Cucumber's DataTable and Scenario Outline with Examples tables. It transforms the tabular data provided in the feature files into Java objects (e.g., List<Map<String, String≫ or custom POJOs) that can be passed directly to the service layer, enabling data-driven testing at its most effective [5].

Layer 4: The Gherkin Feature Files Layer

- This is the topmost and most visible layer, consisting entirely of .feature files written in the Gherkin language. These files are purposefully devoid of any technical implementation details and are written in a domain-specific language accessible to business stakeholders, product owners, manual testers, and developers alike [3], [6].
- Content and Role: Each file describes a software feature and contains multiple scenarios and scenario outlines that define the expected behavior of the system using real-world examples. The language focuses on user goals and system outcomes rather than technical endpoints and request methods. This layer serves as the single source of truth for the system's intended behavior, functioning simultaneously as executable acceptance criteria and as living, up-to-date documentation. The success of this layer is measured solely by its clarity and readability for non-programmers, ensuring that the framework truly embodies the collaborative spirit of BDD [4], [6].

IV. IMPLEMENTATION IN PRACTICE: A WALKTHROUGH

To illustrate the practical application of the layered architecture, consider a common test scenario: "Successfully create a new user." This walkthrough demonstrates how the four layers interact, showcasing the separation of concerns and the flow from a business-readable specification to a technical execution.

International Journal of Advanced Research in Computer Science & Technology (IJARCST)

| ISSN: 2347-8446 | www.ijarcst.org | editor@ijarcst.org |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January-February 2023||

DOI:10.15662/IJARCST.2023.0601003

The process begins at the highest layer of abstraction. A tester or product owner writes the requirement in Gherkin, focusing on the behavior, not the implementation. This is stored in a user_management.feature file.

```gherkin
Scenario Outline: Creating a new user via the API
    Given I have a valid user payload for "<name>" and "<email>"
    When I submit a POST request to the "/users" endpoint
    Then the response status code should be 201
    And the response body should contain the user id and "<email>"

    Examples:
      | name       | email              |
      | John Doe   | john.doe@test.com  |
      | Jane Smith | jane.smith@test.com |
```

Figure 2. Gherkin (Layer 4)

This scenario outline is parsed, and Cucumber matches each step to its corresponding method in the step definitions class. The step definitions act as the glue, extracting the data from the Gherkin step and delegating the actual API interaction to the service layer.

```java
public class UserStepDefinitions {
    private User testUser;
    private Response apiResponse;

    @Given("I have a valid user payload for {string} and {string}")
    public void create_user_payload(String name, String email) {
        testUser = new User(name, email); // Uses Layer 1 POJO
    }

    @When("I submit a POST request to the {string} endpoint")
    public void submit_post_request(String endpoint) {
        // Calls Layer 2 Service
        apiResponse = UserAPIService.createUser(testUser);
    }

    @Then("the response status code should be {int}")
    public void verify_response_status(int expectedStatusCode) {
        apiResponse.then().statusCode(expectedStatusCode);
    }
}
```

Figure 3. Step Definitions (Layer 3)

The UserAPIService in the service layer (Layer 2) contains the actual REST Assured code. It uses the core RequestSpecBuilder from Layer 1 to ensure the request is properly configured and executes the call.

```java
public class UserAPIService {
    public static Response createUser(User userPayload) {
        return given()
                .spec(RequestSpecBuilder.getRequestSpec()) // Layer 1
                .body(userPayload) // Layer 1 POJO serialized to JSON
                .when()
                .post("/users")
                .then()
                .extract()
                .response();
    }
}
```

Figure 4. Service Layer (Layer 2)

TABLE III. LAYER INTERACTION FOR THE "CREATE USER" FLOW

Layer	Component	Action
4. Gherkin	user_management.feature	Defines the business behavior and test data.
3. Step	UserStepDefinitions	Translates

| ISSN: 2347-8446 | www.ijarcst.org | editor@ijarcst.org |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January-February 2023||

DOI:10.15662/IJARCST.2023.0601003

Definitions		Gherkin steps; prepares data and makes service calls.
2. Service Layer	UserAPIService	Encapsulates the API endpoint logic using REST Assured.
1. Core Framework	RequestSpecBuilder, User POJO	Provides base request config and data models.

V. ADVANCED CONSIDERATIONS FOR A PRODUCTION-READY FRAMEWORK

While the core layered architecture provides a solid foundation, deploying a framework into a continuous integration/continuous deployment (CI/CD) pipeline demands addressing advanced considerations that ensure reliability, efficiency, and robustness in a production environment.

A critical enhancement is externalized configuration management. Hardcoding environment-specific variables (e.g., base URLs, credentials) is a fatal flaw for portability. A production-grade framework must read these parameters from external configuration files (e.g., .properties, .yaml) or environment variables. This allows the same test codebase to execute seamlessly against different environments—such as development, staging, and production—without any modification, simply by switching the active profile [1], [5]. Furthermore, secure handling of secrets, such as API keys and passwords, is paramount. These should never be stored in version control but instead be injected at runtime through secure vaults or CI/CD pipeline variables.

The other critical factor is advanced test data management. The tests are to be isolated and idempotent and thus they should be able to execute on their own and can be reused over and over again without leading to failure because of collision of data. One such approach is the programmatic generation of prerequisite data in the Before hook and the careful cleanup of artifacts (e.g. deletion of test users) in the After or AfterAll hook to ensure a clean test environment [2]. In the case of data-driven testing, the creation of unique and random data at each test execution with the help of such libraries as Java Faker avoids conflicts and guarantees the reliability of the tests.

Last but not least, the framework has the most value to offer when integrated into a CI/CD pipeline. Such tools as Jenkins, GitHub Actions or GitLab CI can be specified to automatically run the test suite on events such as a pull request or a nightly build. Organizing the framework to run tests concurrently, taking advantage of such tools as parallel execution offered by Cucumber or parallel execution offered by JUnit, makes the feedback time significantly smaller, making the pipeline efficient and agile [4]. This is a continuous automated testing that offers a vital safety net to deployment, where code integrations are not permitted to affect current functionality of the API.

REFERENCES

[1] B. G. Wolde and A. S. Boltana, “REST API composition for effectively testing the Cloud,” Journal of Applied Research and Technology, vol. 19, no. 6, pp. 676–693, Dec. 2021, doi: 10.22201/icat.24486736e.2021.19.6.1653.
[2] N. Palani, Software Automation Testing Secrets Revealed: Revised Edition-Part 1. Delhi, India: Educreation Publishing, 2017.
[3] V. Österholm, “Overview of Behaviour-Driven Development tools for web applications,” M.S. thesis, Dept. Comput. Sci., KTH Royal Institute of Technology, Stockholm, Sweden, 2021. [Online]. Available:
[4] B. G. Wolde and A. S. Boltana, “Behavior-driven quality first Agile testing for cloud service,” International Journal of Software Engineering & Applications, vol. 12, no. 1, pp. 13–27, Jan. 2021, doi: 10.5121/ijsea.2021.12102.
[5] P. A. Chaubal, Mastering Behavior-Driven Development Using Cucumber: Practice and Implement Page Object Design Pattern, Test Suites in Cucumber, POM TestNG Integration, Cucumber Reports, and Work with Selenium Grid. India: BPB Publications, 2021.

| ISSN: 2347-8446 | www.ijarcst.org | editor@ijarcst.org |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January-February 2023||

DOI:10.15662/IJARCST.2023.0601003

[6] I. Nečas, "BDD as a specification and QA instrument," M.S. thesis, Faculty of Informatics, Masaryk University, Brno, Czech Republic, 2011.
[7] R. B. Bahaweres, E. Oktaviani, L. K. Wardhani, I. Hermadi, A. Suroso, I. P. Solihin, and Y. Arkeman, "Behavior-driven development (BDD) Cucumber Katalon for Automation GUI testing case CURA and Swag Labs," in 2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS), Jakarta, Indonesia.
[8] N. Li, A. Escalona, and T. Kamal, "Skyfire: Model-based testing with cucumber," in 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), Chicago, IL, USA, Apr. 2016, pp. 393–400, doi: 10.1109/ICST.2016.42.