# Comparison Report: Testing APIs using BDD/Gherkin User Stories and Contract Testing

## Overview

A report that compares two different approaches to API testing using BDD and Contract Testing principles. Both strategies aim to test API endpoints documented in Swagger/OpenAPI, leveraging a domain-specific language (DSL) for writing test cases in a natural-language style. This allows for requirements, or more specifically, the understanding of the requirements, to evolve and change throughout the life of a project and be documented in a manner that encourages collaboration between developers, QA, and non-technical or business stakeholders.

The approaches are:

1. **Typescript/Cucumber/Cypress**
2. **C#/SpecFlow/Playwright**

## Approach #1: Typescript/Cucumber/Cypress

**Pros:**

* **Natural Language Test Authoring (Gherkin):**
  + Utilizes **Cucumber** for writing BDD-style test cases in Gherkin, making tests more understandable to non-technical stakeholders.
* **Strong Type Safety (Typescript):**
  + Typescript adds static typing to JavaScript, reducing runtime errors and improving code quality.
* **Mature Testing Ecosystem (Cypress):**
  + **Cypress** is widely used for both frontend and API testing, offering a **fast execution** environment and **real-time test monitoring**.
* **Comprehensive Testing Support:**
  + Supports a wide range of testing scenarios (UI, API, and end-to-end tests) within one framework.
* **Easy Setup and Integration:**
  + Cypress has a **simple setup** for API testing with good support for testing against **Swagger/OpenAPI** documentation.
* **Rich Debugging Capabilities:**
  + Cypress provides **automatic snapshots** and a rich debugging environment that makes it easy to trace issues in API calls.

**Cons:**

* **Lesser Focus on API-First Development:**
  + While Cypress excels in UI testing, its primary focus is not on API testing, and it may lack **dedicated features** for API-first testing strategies like contract testing.
* **Scalability Concerns:**
  + Cypress is a **browser-centric tool**, which may become less effective or inefficient when scaling to more complex or **high-volume API test suites**.
* **Limited Parallelization:**
  + Test execution in Cypress is inherently slower when dealing with **large test suites** due to its limited support for parallelization across multiple machines.

## Approach #2: C#/SpecFlow/Playwright

**Pros:**

* **Natural Language Test Authoring (Gherkin):**
  + **SpecFlow** offers Gherkin syntax for writing BDD-style test cases, providing a **business-readable language** that bridges the gap between technical and non-technical teams.
* **Strong Type Safety and Performance (C#):**
  + C# provides **strongly-typed object models**, making the test implementation more robust and less error-prone.
  + C# is **performance-oriented**, making it suitable for **large-scale API testing** where speed and efficiency are key.
* **Powerful API Testing Features (Playwright):**
  + **Playwright** supports both frontend and API testing with **robust capabilities** for handling network requests and API validation.
  + Excellent support for testing RESTful APIs as well as **contract testing** strategies.
* **Parallel Test Execution:**
  + Playwright and SpecFlow both provide **native support for parallelization**, improving test execution time for large test suites.
* **Better Integration with Swagger/OpenAPI:**
  + Playwright, combined with C#, has more mature libraries for **Swagger/OpenAPI integration**, allowing for easier **contract validation** and **automated schema checks**.

**Cons:**

* **Higher Learning Curve (C# and SpecFlow):**
  + SpecFlow and C# may have a **steeper learning curve**, especially for teams familiar with JavaScript/Typescript ecosystems.
* **Longer Setup Time:**
  + Initial setup and configuration of Playwright and SpecFlow can be more **time-consuming** compared to Cypress.
* **Less Community Support:**
  + While **SpecFlow** and **Playwright** are powerful, their **community** and resources are somewhat smaller compared to the **Cucumber/Cypress** ecosystem.

## Summary of Key Considerations

| **Factor** | **Typescript/Cucumber/Cypress** | **C#/SpecFlow/Playwright** |
| --- | --- | --- |
| **Test Case Authoring** | Gherkin via Cucumber; readable but less API-focused | Gherkin via SpecFlow; strong domain-driven design support |
| **Language** | Typescript; less strict but more popular | C#; strict typing and high performance |
| **API Testing Focus** | Limited, primarily UI-centric | Strong API and contract testing support |
| **Swagger/OpenAPI Integration** | Basic support via Cypress plugins | Advanced, integrated support |
| **Parallel Test Execution** | Limited | Strong support |
| **Performance** | Generally fast | Generally fast |
| **Community and Ecosystem** | Large ecosystem, many plugins | SpecFlow - Large and mature  Playwright - Smaller but growing community |
| **Ease of Setup** | Faster setup | Requires more configuration |
| **Debugging and Reporting** | Strong (automatic snapshots, real-time monitoring) | Strong (detailed logs and tracing) |
| |  | | --- | | **Learning Curve** |  |  | | --- | |  |  |  | | --- | |  | | **Low** – Easy to learn for teams familiar with JavaScript | **High** – Steeper curve due to C# and more complex tooling |

## Conclusion

Both approaches have distinct advantages and limitations.

**Typescript/Cucumber/Cypress** is a solid choice for teams prioritizing **ease of use and quick setup,** especially if they have existing **JavaScript or TypeScript expertise.** It's well-suited for scenarios where **UI** and **basic API testing** are the primary focus**.** However, it may be less ideal for teams heavily focused on **API-first strategies** or **contract testing.**

**C#/SpecFlow/Playwright** is a more robust and scalable solution for **comprehensive API testing.** It's particularly well-suited for teams with a strong **.NET background** and those prioritizing **API-first development** and **contract testing.** While it offers strong integration with **OpenAPI/Swagger** and advanced testing capabilities, it requires a **steeper learning curve** and more initial setup.

### Typescript/Cucumber/Cypress:

* Ideal for teams prioritizing **ease of use** and **quick setup**.
* **Low learning curve**, especially for teams familiar with JavaScript.
* **Large Community and Ecosystem:** Extensive support and resources for Typescript, Cucumber, and Cypress.
* **Modern Framework:** Cypress is a modern testing framework with a user-friendly API and powerful features.
* Strong choice for **UI and basic API testing** in a JavaScript-based ecosystem.
* Suitable for projects where **fast feedback** and **lightweight API testing** are needed.
* Less appropriate, but can be adapted, for teams focusing on **API-first strategies** or **contract testing**.

### C#/SpecFlow/Playwright:

* Ideal for teams with a strong .NET background.
* Better for teams focusing on **API-first development** and **contract testing**.
* **Mature Technology Stack:** C# and .NET are well-established technologies with a strong foundation.
* **Higher learning curve**, especially for those unfamiliar with C# and SpecFlow tooling.
* **Potential for Boilerplate Code:** C# can sometimes require more verbose syntax compared to Typescript.
* **Powerful Automation Tool:** Playwright is a powerful automation library that supports multiple browsers and platforms.
* **Robust Testing Framework:** SpecFlow is a mature BDD framework with strong support for .NET.
* Offers strong integration with **OpenAPI/Swagger** for thorough API validation.
* Highly scalable for **large, complex API test suites**.
* Requires more setup and has a **steeper learning curve** but provides **robust performance** and **advanced testing capabilities**.

**Ultimately,** the best approach depends on the team's specific needs, skillset, and project requirements. Consider factors such as **learning curve**, **team expertise**, and **long-term maintenance** when making a decision.

## Benefits of Testing APIs using BDD/Gherkin Basic Smoke Tests

Basic smoke tests validate the core functionality of an API to ensure it is operational. Writing these tests in BDD/Gherkin format offers several advantages, especially in terms of simplicity and accessibility.

**Key Benefits:**

* **Clear and Concise Test Cases:**
  + BDD/Gherkin uses plain language that is easily understandable by technical and non-technical stakeholders alike.
  + Scenarios are described in a "Given-When-Then" structure, making the purpose of each test case clear.
* **Focus on Core Functionality:**
  + Smoke tests ensure that the most critical aspects of each API endpoint function correctly.
  + These tests provide immediate feedback on the operational status of the API.
* **Simplicity:**
  + Each test is designed to validate a single, basic operation (e.g., status codes and basic response structure).
  + The format avoids technical jargon, focusing on what the API does (it’s behaviour) rather than implementation details.
* **Enhanced Collaboration:**
  + BDD/Gherkin scenarios can be authored and reviewed by product owners, QA teams, developers, and other stakeholders.
  + Encourages collaboration and shared understanding of the API’s functionality.
* **Ease of Maintenance:**
  + BDD/Gherkin tests are written at a high level, minimizing the need for complex code changes when the API evolves.
  + Simple scenarios ensure that updates are easy to implement.
* **Fast Feedback Loop:**
  + Smoke tests execute quickly, providing immediate insights into whether the API is fundamentally operational after deployment.

## Why Use BDD/Gherkin for Smoke Tests?

* **Enhanced Test Reusability:**
  + BDD/Gherkin tests can be reused across different test environments and platforms.
* **Ensures API Operational Readiness:**
  + Verifies that all critical endpoints are functional before proceeding to more extensive testing.
* **Promotes Team Collaboration:**
  + The natural language format fosters communication and agreement on what constitutes a "healthy" API.
* **Streamlined Test Execution:**
  + Lightweight tests execute quickly and efficiently, ideal for inclusion in CI/CD pipelines.
* **Universal Accessibility:**
  + Stakeholders from any background can read, understand, and contribute to the test definitions.

The simplicity, combined with the emphasis on core functionality, makes smoke tests in BDD/Gherkin format, an ideal starting point for validating APIs.

## Example Test Approach

## API Endpoints documented in Swagger/OpenAPI

## Contract Testing using BDD/Gherkin Test Cases

This section presents an example test approach to an API documented using Swagger/OpenAPI, with corresponding test cases written in declarative BDD/Gherkin style.

The API has three endpoints:

1. Alive (GET): Health checker of the API.
2. Parse-dynamic-string (GET): Outputs a computed string from an input token.
3. Parse-date-token (GET): Outputs a computed date from an input token date.

All test cases are written in declarative Gherkin style, focusing on contract testing to ensure the API's functionality, response structure, and status codes meet the specification.

## Swagger/OpenAPI Example (Simplified)

openapi: 3.0.0

info:

title: Example API

version: 1.0.0

paths:

/alive:

get:

summary: Health check endpoint

responses:

'200':

description: API is alive and functioning

content:

application/json:

schema:

type: object

properties:

status:

type: string

example: "alive"

/Parse-dynamic-string:

get:

summary: Parses a dynamic string token

parameters:

- name: tokenString

in: query

required: true

schema:

type: string

responses:

'200':

description: Successfully parsed string token

content:

application/json:

schema:

type: object

properties:

ParsedToken:

type: string

example: "generatedstring"

'400':

description: Invalid input or parsing error

content:

application/json:

schema:

type: object

properties:

Error:

type: string

example: "Invalid token"

/Parse-date-token:

get:

summary: Parses a date token

parameters:

- name: tokenString

in: query

required: true

schema:

type: string

format: date

responses:

'200':

description: Successfully parsed date token

content:

application/json:

schema:

type: object

properties:

ParsedToken:

type: string

example: "generateddate"

'400':

description: Invalid input or parsing error

content:

application/json:

schema:

type: object

properties:

Error:

type: string

example: "Invalid date token"

## BDD/Gherkin User Stories for API Testing

User stories are written in a **declarative style**, with a focus on the **contract testing** approach to ensure that the API adheres to its contract (as defined by Swagger/OpenAPI) by testing each endpoint’s response against the expected schema and behaviour.

**Scenario 1: Check if the API is alive**

Feature: API Health Check Endpoint

  As an API consumer

I want to receive a confirmation that the API is alive

So I can verify that the API is operational

Scenario: API responds successfully to a health check

Given the API is available

When a GET request is made to the /alive endpoint

Then the API response should return a status code of 200

And the response body should contain "status" with the value "alive"

**Scenario 2: Parse a dynamic token into a string**

Feature: Parse Dynamic String Token Endpoint

  As an API consumer

  I want to ensure that valid tokens are correctly parsed into dynamic strings

  So that I can use the API for dynamic string token parsing

Scenario: Successfully parse a valid dynamic string token

Given a valid dynamic token "[exampleDynamicToken]" is provided

When a GET request is made to the /Parse-dynamic-string endpoint with the token

Then the API response should return a status code of 200

And the response body should contain "ParsedToken" with the value "generatedstring"

Scenario: Fail to parse an invalid dynamic string token

Given an invalid dynamic token "invalidToken" is provided

When a GET request is made to the /Parse-dynamic-string endpoint with the token

Then the API response should return a status code of 400

And the response body should contain "Error" with the value "Invalid token"

**Scenario 3: Parse a token into a date string**

Feature: Parse Date Token Endpoint

  As an API consumer

  I want to ensure that valid date tokens are correctly parsed into formatted dates

  So that I can use the API for date token parsing

Scenario: Successfully parse a valid date token

Given a valid date token "[exampleDateToken]" is provided

When a GET request is made to the /Parse-date-token endpoint with the date token

Then the API response should return a status code of 200

And the response body should contain "ParsedToken" with the value "generateddate"

And the "ParsedToken" should be a valid date in the format "yyyy-MM-dd HH:mm:ssZ"

Scenario: Fail to parse an invalid date token

Given an invalid date token "invalidDate" is provided

When a GET request is made to the /Parse-date-token endpoint with the date token

Then the API response should return a status code of 400

And the response body should contain "Error" with the value "Invalid date token"

### Explanation of Declarative Style:

* **Given**: States a condition or pre-existing context (e.g., "a valid token is provided").
* **When**: Describes the event/action that triggers the test (e.g., "a GET request is made").
* **Then**: Defines the expected outcome in a straightforward manner (e.g., "the API response should return a status code of 200").

The focus is on what the system does, not how it does it, keeping the scenarios high-level, readable, and easy for non-technical stakeholders to understand.

## Pseudo Code Summary for Both Approaches (Code-Base Agnostic)

The following pseudo-code summarizes the structure of contract testing in the code base. It represents the test implementations for both the approaches

**Typescript/Cucumber/Cypress** and **C#/SpecFlow/Playwright**.

It is agnostic to the language and/or the testing framework used, focusing instead on the general structure.

### Test Implementation (Pseudo Code)

**Test Setup:**

BeforeAll:

Initialize API base URL from Swagger/OpenAPI documentation

Initialize HTTP client

BeforeEach:

Prepare request headers and query parameters for each scenario

**Test for Health Check Endpoint:**

Test "API is alive":

Send GET request to "/alive"

Assert response status is 200

Assert response body contains "API is alive"

**Test for Dynamic String Token Parsing:**

Test "Token dynamic string parser":

Set query parameter "token" to "myToken123"

Send GET request to "/token/dynamicStringParser"

Assert response status is 200

Parse response body as JSON

Assert JSON response contains key "ParsedToken"

Assert "ParsedToken" equals "generatedstring"

**Test for Date Token Parsing:**

Test "Token date parser":

Set query parameter "token" to "tokenDate"

Send GET request to "/token/dateParser"

Assert response status is 200

Parse response body as JSON

Assert JSON response contains key "ParsedToken"

Assert "ParsedToken" equals "2023-01-01"

**Test Teardown:**

AfterEach:

Reset HTTP client session if necessary

AfterAll:

Clean up resources or close API connections

**Key Testing Concepts Addressed:**

* **Declarative Gherkin Style**: The test scenarios are written in a clear, business-readable format, focusing on the desired outcomes rather than implementation specifics.
* **Contract Testing**: Each test validates the API’s contract by checking that the response status and body comply with the expected output as defined in the Swagger/OpenAPI documentation.
* **Test Coverage**: All endpoints (/alive, /token/dynamicStringParser, and /token/dateParser) are tested for both **functional correctness** and **contract validation**.

This pseudo code and Gherkin examples can be adapted to either the **Typescript/Cucumber/Cypress** or **C#/SpecFlow/Playwright** implementation while adhering to BDD and contract testing principles.