API Testing Approach

The API layer of any application is one of the most crucial software components. It is the channel which connects client to server (or one microservice to another), drives business processes, and provides the services which give value to users.
A customer-facing public API that is exposed to end-users becomes a product in itself. If it breaks, it puts at risk, not just a single application, but an entire chain of business processes built around it.
Mike Cohn's famous Test Pyramid places API tests at the service level (integration), which suggests that around 20% or more of all of our tests should focus on APIs (the exact percentage is less important and varies based on our needs).
Once we have a solid foundation of unit tests which cover individual functions, API tests provide higher reliability covering an interface closer to the user, yet without the brittleness of UI tests.
API tests are fast, give high ROI, and simplify the validation of business logic, security, compliance, and other aspects of the application. In cases where the API is a public one, providing end-users programmatic access to our application or services, API tests effectively become end-to-end tests and should cover a complete user story.

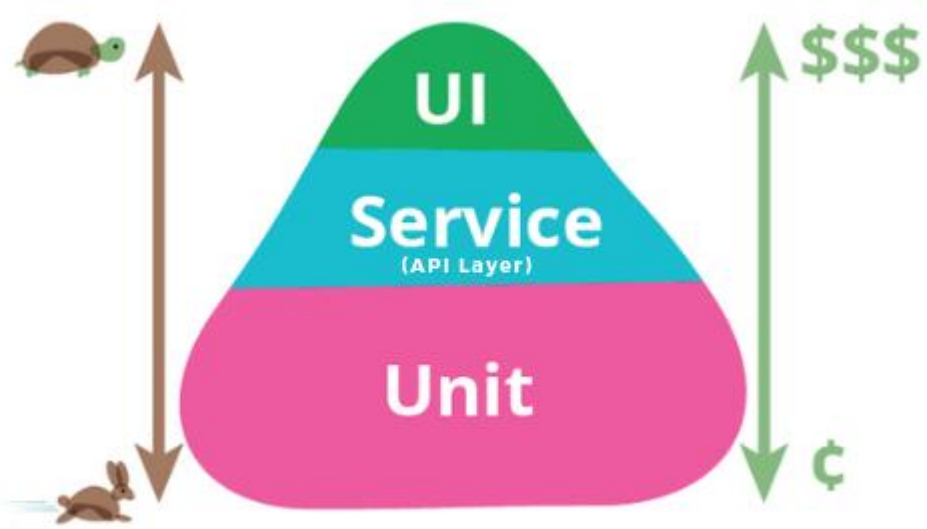

Image source: Martin Fowler
So the importance of API testing is obvious. Several methods and resources help with **HOW** to test APIs — manual testing, automated testing, test environments, tools, libraries, and frameworks. However, regardless of what you will use — *Postman, supertest, pytest, JMeter, mocha, Jasmine, RestAssured*, or any other tools of the trade — before coming up with any test method you need to determine **what to test**…

API test strategy

The test strategy is the high-level description of the test requirements from which a detailed test plan can later be derived, specifying individual test scenarios and test cases. Our first concern is **functional testing** — ensuring that the API functions correctly.
The main objectives in functional testing of the API are:

- to ensure that the implementation is working correctly as expected — no bugs!
- to ensure that the implementation is working as specified according to the requirements specification (which later on becomes our API documentation).
- to prevent regressions between code merges and releases.

API as a contract — first, check the spec!

An API is essentially a contract between the client and the server or between two applications. Before any implementation test can begin, it is important to make sure that the contract is correct. That can be done first by inspecting the spec (or the service contract itself, for example a Swagger interface or OpenAPI reference)

and making sure that **endpoints are correctly named,** that **resources and their types correctly reflect the object model,** that **there is no missing functionality** or **duplicate functionality,** and that **relationships between resources are reflected in the API correctly.**
The guidelines above are applicable to any API, but for simplicity, in this post, we assume the most widely used Web API architecture — **REST over HTTP**. If your API is designed as a truly RESTful API, it is important to **check that the REST contract is a valid one**, including all HTTP REST semantics, conventions, and principles (here, here, and here).

If this is a customer-facing public API, this might be your last chance to ensure that all contract requirements are met, because once the API is published and in use, any changes you make might break customers' code.

(Sure, you can publish a new version of the API someday (e.g., /api/v2/), but even then backward compatibility might still be a requirement).

So, what aspects of the API should we test?

Now that we have validated the API contract, we are ready to think of what to test. Whether you're thinking of test automation or manual testing, our functional test cases have the same **test actions**, are part of wider **test scenario categories,** and belong to three kinds of **test flows.**

API test actions

Each test is comprised of test actions. These are the individual actions a test needs to take per API test flow. For each API request, the test would need to take the following actions:

1. Verify correct HTTP status code. For example, creating a resource should return 201 CREATED and unpermitted requests should return 403 FORBIDDEN, etc.

2. Verify response payload. Check valid JSON body and correct field names, types, and values — including in error responses.

3. Verify response headers. HTTP server headers have implications on both security and performance.

4. Verify correct application state. This is optional and applies mainly to manual testing, or when a UI or another interface can be easily inspected.

5. Verify basic performance sanity. If an operation was completed successfully but took an unreasonable amount of time, the test fails.

Test scenario categories

Our test cases fall into the following general *test scenario* groups:

- Basic positive tests (happy paths)
- Extended positive testing with optional parameters
- Negative testing with valid input
- Negative testing with invalid input
- Destructive testing
- Security, authorization, and permission tests (which are out of the scope of this post)

Happy path tests check basic functionality and the acceptance criteria of the API. We later extend positive tests to include optional parameters and extra functionality. The next group of tests is **negative testing** where we expect the application to gracefully handle problem scenarios with both valid user input (for example, trying to add an existing username) and invalid user input (trying to add a username which is null). **Destructive**

testing is a deeper form of negative testing where we intentionally attempt to break the API to check its robustness (for example, sending a huge payload body in an attempt to overflow the system).

Test flows

Let's distinguish between three kinds of test flows which comprise our test plan:
1. Testing requests in isolation – Executing a single API request and checking the response accordingly. Such basic tests are the minimal building blocks we should start with, and there's no reason to continue testing if these tests fail.
2. Multi-step workflow with several requests – Testing a series of requests which are common user actions, since some requests can rely on other ones. For example, we execute a POST request that creates a resource and returns an auto-generated identifier in its response. We then use this identifier to check if this resource is present in the list of elements received by a GET request. Then we use a PATCH endpoint to update new data, and we again invoke a GET request to validate the new data. Finally, we DELETE that resource and use GET again to verify it no longer exists.
3. Combined API and web UI tests – This is mostly relevant to manual testing, where we want to ensure data integrity and consistency between the UI and API.

We execute requests via the API and verify the actions through the web app UI and vice versa. The purpose of these integrity test flows is to ensure that although the resources are affected via different mechanisms the system still maintains expected integrity and consistent flow.

An API example and a test matrix

We can now express everything as a matrix that can be used to write a detailed test plan (for test automation or manual tests).
Let's assume a subset of our API is the */users* endpoint, which includes the following API calls:

API Call	Action
GET /users	List all users
GET /users?name={username}	Get user by username
GET /users/{id}	Get user by ID
GET /users/{id}/configurations	Get all configurations for user
POST /users/{id}/configurations	Create a new configuration for user
DELETE /users/{id}/configurations/{id}	Delete configuration for user
PATCH /users/{id}/configuration/{id}	Update configuration for user

Where *{id}* is a UUID, and all GET endpoints allow optional query parameters *filter*, *sort*, *skip* and *limit* for filtering, sorting, and pagination.

#	Test Scenario Category	Test Action Category	Test Action Description
1	**Basic positive tests (happy paths)**		
	Execute API call with **valid required parameters**	Validate status code:	1. All requests should return 2XX HTTP status code

			2. Returned status code is according to spec: – 200 OK for GET requests – 201 for POST or PUT requests creating a new resource – 200, 202, or 204 for a DELETE operation and so on
		Validate payload:	1. Response is a well-formed JSON object 2. Response structure is according to data model (schema validation: field **names** and field **types** are as expected, including nested objects; field **values** are as expected; non-nullable fields are not null, etc.)
		Validate state:	1. For GET requests, verify there is NO STATE CHANGE in the system (idempotence) 2. For POST, DELETE, PATCH, PUT operations – Ensure action has been performed correctly in the system by: – Performing appropriate GET request and inspecting response – Refreshing the UI in the web application and verifying new state (only applicable to manual testing)
		Validate headers:	Verify that HTTP headers are as expected, including *content-type,* *connection,* *cache-control,* *expires,* *access-control-allow-origin,* *keep-alive,* *HSTS,* and other standard header fields – according to spec. Verify that information is NOT leaked via headers (e.g. *X-Powered-By* header is not sent to user).
		Performance sanity:	Response is received in a timely manner (within reasonable expected time) — as defined in the test plan.
2	**Positive + optional parameters**		
	Execute API call with **valid required parameters AND valid optional** parameters	Validate status code:	As in #1

	Run same tests as in #1, this time including the endpoint's optional parameters (e.g., filter, sort, limit, skip, etc.)		
		Validate payload:	Verify response structure and content as in #1. In addition, check the following parameters: – filter: ensure the response is filtered on the specified value. – sort: specify field on which to sort, test ascending and descending options. Ensure the response is sorted according to selected field and sort direction. – skip: ensure the specified number of results from the start of the dataset is skipped – limit: ensure dataset size is bounded by specified limit. – limit + skip: Test pagination Check combinations of all optional fields (fields + sort + limit + skip) and verify expected response.
		Validate state:	As in #1
		Validate headers:	As in #1
		Performance sanity:	As in #1
3	**Negative testing – valid input**		
	Execute API calls with **valid input** that attempts illegal operations. i.e.: – Attempting to create a resource with a name that already exists (e.g., user configuration with the same name) – Attempting to delete a resource that doesn't exist (e.g., user configuration with no such ID) – Attempting to update a resource with illegal valid data (e.g., rename a configuration to an existing name)	Validate status code:	1. Verify that an erroneous HTTP status code is sent (NOT 2XX) 2. Verify that the HTTP status code is in accordance with error case as defined in spec

	– Attempting illegal operation (e.g., delete a user configuration without permission.) And so forth.		
		Validate payload:	1. Verify that error response is received 2. Verify that error format is according to spec. e.g., error is a valid JSON object or a plain string (as defined in spec) 3. Verify that there is a clear, descriptive error message/description field 4. Verify error description is correct for this error case and in accordance with spec
		Validate headers:	As in #1
		Performance sanity:	Ensure error is received in a timely manner (within reasonable expected time)
4	**Negative testing – invalid input**		
	Execute API calls with invalid input, e.g.: – Missing or invalid authorization token – Missing required parameters – Invalid value for endpoint parameters, e.g.: – Invalid UUID in path or query parameters – Payload with invalid model (violates schema) – Payload with incomplete model (missing fields or required nested entities) – Invalid values in nested entity fields – Invalid values in HTTP headers – Unsupported methods for endpoints And so on.	Validate status code:	As in #1
		Validate payload:	As in #1
		Validate headers:	As in #1

		Performance sanity:	As in #1
5	**Destructive testing**		
	Intentionally attempt to fail the API to check its robustness: Malformed content in request Wrong content-type in payload Content with wrong structure Overflow parameter values. E.g.: – Attempt to create a user configuration with a title longer than 200 characters – Attempt to GET a user with invalid UUID which is 1000 characters long – Overflow payload – huge JSON in request body Boundary value testing Empty payloads Empty sub-objects in payload Illegal characters in parameters or payload Using incorrect HTTP headers (e.g. Content-Type) Small concurrency tests – concurrent API calls that write to the same resources (DELETE + PATCH, etc.) Other exploratory testing	Validate status code:	As in #3. API should fail gracefully.
		Validate payload: Validate headers:	As in #3. API should fail gracefully. As in #3. API should fail gracefully.
		Performance sanity:	As in #3. API should fail gracefully.

Test cases derived from the table above should cover different **test flows** according to our needs, resources, and priorities.

Going Beyond Functional Testing

Following the test matrix above should generate enough test cases to keep us busy for a while and provide good functional coverage of the API. Passing all functional tests implies a good level of maturity for an API, but it is not enough to ensure high quality and reliability of the API.
In the next post in this series we will cover the following **non-functional test approaches** which are essential for API quality:

Security and Authorization

- Check that the API is designed according to correct security principles: deny-by-default, fail securely, least privilege principle, reject all illegal inputs, etc.
    - Positive: ensure API responds to correct authorization via all agreed auth methods – Bearer token, cookies, digest, etc. – as defined in spec
    - Negative: ensure API refuses all unauthorized calls
- Role Permissions: ensure that specific endpoints are exposed to user based on role. API should refuse calls to endpoints which are not permitted for user's role
- Protocol: check HTTP/HTTPS according to spec
- Data leaks: ensure that internal data representations that are desired to stay internal do not leak outside to the public API in the response payloads
- Rate limiting, throttling, and access control policies

Performance

- Check API response time, latency, TTFB/TTLB in various scenarios (in isolation and under load)

Load Tests (positive), Stress Tests (negative)

- Find capacity limit points and ensure the system performs as expected under load, and fails gracefully under stress

Usability Tests

- For public APIs: a manual "Product"-level test going through the entire developer journey from documentation, login, authentication, code examples, etc. to ensure the usability of the API for users without prior knowledge of our system.