**Table of Contents**

[1 INTRODUCTION 1](#_Toc96688696)

[2 SCOPE 2](#_Toc96688697)

[3 QUALITY OBJECTIVES 2](#_Toc96688698)

[3.1 Primary Objectives 2](#_Toc96688699)

[3.2 Secondary Objectives 2](#_Toc96688700)

[4 API TEST APPROACH 3](#_Toc96688701)

[4.2 Test flows 4](#_Toc96688702)

4.3 Test Scenario Categories……………………………………………….. 4

**Test Plan**

**Project “Sample”**

Document Revision History

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Date | Version | Description | Author | Reviewer | Approver |
| 25.02 | 0.1 | Test plan was created | Arvind Akula |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# INTRODUCTION

The API Test Plan has been created to facilitate communication within the team members. This document describes approaches and methodologies that will apply to the unit, API testing of the "http://www.topmarques.co.uk

# SCOPE

The document mainly targets the API testing and validating data in output as per Requirements Specifications provided by Client.

2.1 Functions to be tested. x GUI

x Search and Filters Logic x Performance

2.2 Functions not to be tested.

1. Not other than mentioned above in section 2.1

# QUALITY OBJECTIVES

## Primary Objectives

A primary objective of testing is to: assure that the system meets the full requirements, including quality requirements (functional and non-functional requirements) and fit metrics for each quality requirement and satisfies the use case scenarios and maintain the quality of the product. At the end of the project development cycle, the user should find that the project has met or exceeded all their expectations as detailed in the requirements.

## Secondary Objectives

The secondary objectives of testing will be to: identify and expose all issues and associated risks, communicate all known issues to the project team, and ensure that all issues are addressed in an appropriate matter before release. As an objective, this requires careful and methodical testing of the application to first ensure all areas of the system are scrutinized and, consequently, all issues (bugs) found are dealt with appropriately.

# API TEST APPROACH

The approach, that used, is Analytical therefore, in accordance to requirements-based strategy, where an analysis of the requirements specification forms the basis for planning, estimating and designing tests. Test cases will be created during exploratory testing. All test types are determined in Test Strategy.

Team also must used experience-based testing and error guessing utilize testers' skills and intuition, along with their experience with similar applications or technologies.

The project is using an agile approach, with weekly iterations. At the end of each week the requirements identified for that iteration will be delivered to the team and will be tested.

### 4.1 **API 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.

**API Resources:**

**Description**: Get the details for a test plan.

**Endpoint**: <base\_url>/projects/{project\_id}/plans/{id}.json

**Request**: GET

**Formats supported**: JSON

**Params**:

* project\_id - {Integer} The associated project ID (Required)
* id - {Integer} The test plan ID (Required)

### **Response**

{ .... }

### **Example request**

HTTParty.get('https://api.testlodge.com/v1/account/1/projects/1/plans/1.json')

### **Example Response**

**Response code**: 200

{  
 "id":1,  
  "name":"Plan name",  
 "test\_plan\_identifier":"PLA345",  
 "project\_id":1,  
  "created\_at":"2014-02-26T14:33:58Z",  
  "updated\_at":"2014-02-26T14:33:58Z"  
}

**4.3 Test Scenario Categories:**

|  |  |  |
| --- | --- | --- |
| **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, includingcontent-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-Byheader 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  Run same tests as in #1, this time including the endpoint’s optional parameters (e.g., filter, sort, limit, skip, etc.) |  |  |
|  |  | Validate status code: | As in #1 |
|  |  | 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)  – Attempting illegal operation (e.g., delete a user configuration without permission.)  And so forth. |  |  |
|  |  | 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 |
|  |  | 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. |