API Testing Approach

Components of a REST API request/response pair

A REST API request/response pair can be separated into five components:

1. The **request URI**, in the following form: VERB `https://{instance}[/{team-project}]/_apis[/{area}]/{resource}?api-version={version}`
    - *instance*: The Azure DevOps Services organization or TFS server you're sending the request to. They are structured as follows:
        - Azure DevOps Services: `dev.azure.com/{organization}`
        - TFS: `{server:port}/tfs/{collection}` (the default port is 8080, and the value for collection should be `Default Collection` but can be any collection)
    - *resource path*: The resource path is as follows: `_apis/{area}/{resource}`. For example `_apis/wit/workitems`.
    - *api-version*: Every API request should include an api-version to avoid having your app or service break as APIs evolve. api-versions are in the following format: `{major}.{minor}[-{stage}[.{resource-version}]]`, for example:
        - `api-version=1.0`
        - `api-version=1.2-preview`
        - `api-version=2.0-preview.1`

    > *Note:* area *and* team-project *are optional, depending on the API request. Check out the* *TFS to REST API version mapping matrix* *below to find which REST API versions apply to your version of TFS.*

2. HTTP **request message header** fields:
    - A required HTTP method (also known as an operation or verb), which tells the service what type of operation you are requesting. Azure REST APIs support GET, HEAD, PUT, POST, and PATCH methods.
    - Optional additional header fields, as required by the specified URI and HTTP method. For example, an Authorization header that provides a bearer token containing client authorization information for the request.
3. Optional HTTP **request message body** fields, to support the URI and HTTP operation. For example, POST operations contain MIME-encoded objects that are passed as complex parameters.
    - For POST or PUT operations, the MIME-encoding type for the body should be specified in the Content-type request header as well. Some services require you to use a specific MIME type, such as `application/json`.
4. HTTP **response message header** fields:
    - An HTTP status code, ranging from 2xx success codes to 4xx or 5xx error codes. Alternatively, a service-defined status code may be returned, as indicated in the API documentation.
    - Optional additional header fields, as required to support the request's response, such as a `Content-type` response header.
5. Optional HTTP **response message body** fields:
    - MIME-encoded response objects may be returned in the HTTP response body, such as a response from a GET method that is returning data. Typically, these objects are returned in a structured format such as JSON or XML, as indicated by the `Content-type` response header. For example, when you request an access token from Azure AD, it will be returned in the response body as the `access_token` element, one of several name/value paired objects in a data collection. In this example, a response header of `Content-Type: application/json` is also included.

The natural progression is to test a series of requests that are common actions.

The best design idea here, in my experience, is to keep these isolated, to start, to specific entities. Something like this is what I have in mind:

1. `POST` a new user
2. `GET` new user and validate data
3. `PUT` updated user new data
4. `GET` user and validate new data
5. `DELETE` user
6. `GET` user and validate it doesn't exist

REST services are widely used now a days and hence need to test them properly.

Below are some of the good test cases for REST services

1. API REST Testing involve testing of response message as well as response body.
2. Verification of status code. For Example:
 Code Status
 200 OK
 201 Created

202 Accepted
203 Non-Authoritative information
204 No Content
302 Found
400 Bad Request
401 Unauthorized
404 Not Found
500 Internal Server Error
502 Bad Gateway
503 Service Unavailable

3. Response needs to test based on various inputs
4. Response needs to verify for leaving mandatory fields
5. Tests with unauthorized users.
6. Verify response time of api calls varies considerably.

Example:

Test each endpoint with the methods it supports.

For example, if you have a `/users` endpoint that supports `GET` and `POST` then you have two test cases for requests to that endpoint.

This part can be as simple or complex as you need - if the `GET` requests supports specific parameters you can test them all; use something like J SON Schema to validate the response body; send invalid parameters and things that might 'break' the endpoint; and test that error messages are returned correctly.

A good post on how to write tests for most of these HTTP methods.

Test multi-step workflows

The natural progression is to test a series of requests that are common actions. The best design idea here, in my experience, is to keep these isolated, to start, to specific entities. Something like this is what I have in mind:

1. *`POST` a new user*
2. *`GET` new user and validate data*
3. *`PUT` updated user new data*
4. *`GET` user and validate new data*
5. *`DELETE` user*
6. *`GET` user and validate it doesn't exist*

Keeping the multi-step workflows isolated in their own tests helps reduce interdependence between actions/modules/effects and improves error messages.

Even in the scenario above, I would split the tests into (1, 2), (3, 4), (5, 6) to decrease flakiness, but you get the idea.

Challenges for API Testing

The interesting problems for testers are:

1. To make sure that the test harness varies the parameters of the API calls in such a way that it verifies the functionality as well as expose the failures. It includes exploring boundary condition and assigning common parameters
2. Creating interesting parameter value combinations for calls with two or more parameters
3. Identifying the content under which the API calls have to be made. Which might include setting external environment conditions (peripheral devices, files, etc.) as well as internally stored data that affects the API
4. Sequencing API calls as per the order in which the function will be executed
5. To make the API produce useful results from successive calls.