A Unit Test Guide for Token-Based Authentication

Authors: Danielle Gonzalez (dng2551@rit.edu) & Michael Rath Published: March 2020

Organization

- This guide is a collection of unit/integration test cases for JUnit (Java) implementations of token-based authentication
- Test cases are organized into 5 sections representing unique authentication *features* such as *token manipulation* and *login*
- In each section, test case descriptions are organized by *scenario* (e.g. *authentication failed*)
- A *flow diagram* is also provided for each feature that visualizes the unique combinations of scenario, conditions, and expected outcomes that make up the test cases
- At the end of the guide, unit test smells to avoid are described. These were found to be common in unit tests for Java token authentication in open source software

A Brief Explanation of Token-Based Authentication

When a user submits a login request, applications using token-based authentication use the provided credentials to initialize a **token** object on the server, which is then passed to the authentication mechanism.

If authentication succeeds, the server returns an *authenticated* token back to the client, which is stored & included in the header of subsequent requests to the application so any necessary re-authenticate can occur automatically (until it expires). This way, a user is not repeatedly asked for their credentials as they navigate the application.

An intuitive but detailed explanation of this process can be found here.

Why Should I Write Unit Tests for This?

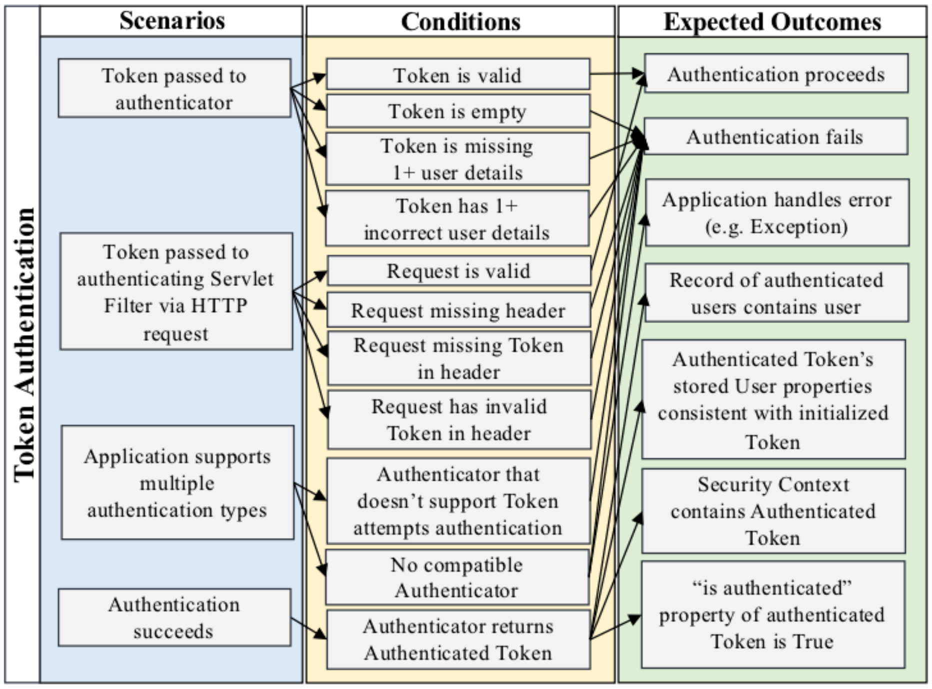
Even with the help of a security API, authentication can be complicated to code (and test!). The implementation requires multiple components to exchange and store information, and their status must be updated and monitored. Further, the authentication *flow* consists of many *behaviors* that are expected to have specific outcomes for different scenarios and conditions.

Unit testing has advantages that other testing approaches cannot fully provide:

- Vulnerabilities and incorrect behavior can be identified *early and often*
- Test methods can act as *documentation* and *proof-of-concepts*
- Internal (non-client-facing) behaviors can be *dynamically evaluated*
- When a test fails, *the root cause* and *responsible code location* can be quickly identified

Tests for Authenticating with Tokens

The authentication process should also be tested to ensure the application correctly handles valid and invalid tokens & servlet requests, performs support checks when multiple authentication mechanisms are enabled, and updates state when authentication is successful.


Token Authentication
Scenarios
Conditions
Expected Outcomes
Token passed to authenticator
Token passed to authenticating Servlet Filter via HTTP request
Application supports multiple authentication types
Authentication succeeds
Token is valid
Token is empty
Token is missing 1+ user details
Token has 1+ incorrect user details
Request is valid
Request missing header
Request missing Token in header
Request has invalid Token in header
Authenticator that doesn't support Token attempts authentication
No compatible Authenticator
Authenticator returns Authenticated Token
Authentication proceeds
Authentication fails
Application handles error (e.g. Exception)
Record of authenticated users contains user
Authenticated Token's stored User properties consistent with initialized Token
Security Context contains Authenticated Token
"is authenticated" property of authenticated Token is True

Scenario: Token passed to authenticator

Authentication should fail when the provided token is:

- Empty
- Missing 1 or more credentials (e.g. username or password)
- Contains 1 or more *incorrect* credentials
- Expired

The authentication process should proceed (check credentials) if the token is valid.

Scenario: Application supports multiple authentication types

Applications supporting multiple authentication types should also ensure that only the correct authenticator can try to authenticate a token.
In this situation, authentication should fail if:

- A non-supporting authenticator attempts to authenticate a token
- The correct authenticator for the token is not found

Scenario: Token is passed to authentication servlet filter via HTTP Request

Applications that use servlet filters for authentication should only attempt to authenticate valid `HTTPServletRequest` requests.

Authentication should fail when the servlet is passed an `HTTPServletRequest` that:

- Does not contain a header
- Does not contain a token in the header
- Contains an invalid token in the header

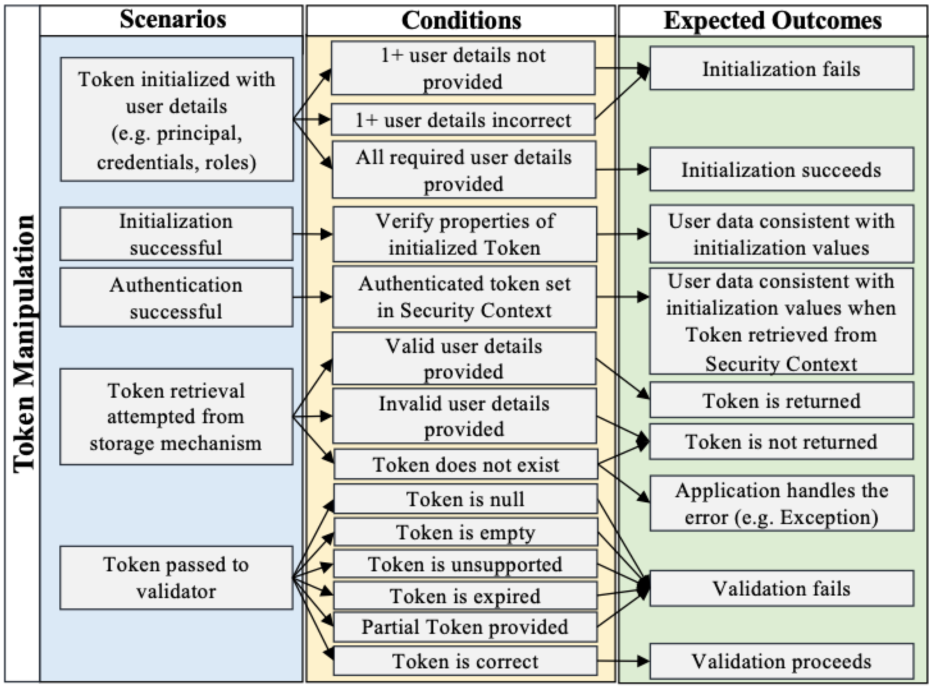
Scenario: Authentication succeeds

After a successful authentication, the token and application's state should be updated.
The following properties should be verified:

- The record of currently authenticated users should contain the user
- Any authentication status "flags", such as an "is authenticated" token property, should be True
- The authentication method returned an authenticated token
- The authenticated token:
    - Is not null
    - Is set within the Security Context
- Any user properties stored in the authenticated token should be consistent with the information used to initialize the pre-authenticated token

Tests for Token Manipulation

Tokens are a core component in this type of authentication: they are used to store, share, and authenticate user information. Thus, it is important to test an application's token initialization, storage, access, and validation behaviors when various types of valid *and* invalid tokens & stored properties are provided.


Token Manipulation
Scenarios
Conditions
Expected Outcomes
Token initialized with user details (e.g. principal, credentials, roles)
1+ user details not provided
1+ user details incorrect
All required user details provided
Initialization fails
Initialization succeeds
Initialization successful
Verify properties of initialized Token
User data consistent with initialization values
Authentication successful
Authenticated token set in Security Context
User data consistent with initialization values when Token retrieved from Security Context
Token retrieval attempted from storage mechanism
Valid user details provided
Invalid user details provided
Token does not exist
Token is returned
Token is not returned
Application handles the error (e.g. Exception)
Token passed to validator
Token is null
Token is empty
Token is unsupported
Token is expired
Partial Token provided
Token is correct
Validation fails
Validation proceeds

Scenario: Token is initialized with user details (e.g. principal, credentials…)

Initialization should fail if 1 or more necessary data (e.g. principal, credentials, role list) are:

- Not provided
- Incorrect

Scenario: Initialization was successful

Once the token is created, tests should verify that the Token's user data properties can be retrieved, and the correct values were stored:

- The principal (username) property should match the value used for initialization
- The password (credentials) property should match the value used for initialization

Scenario: Authentication was successful

The stored properties of the authenticated token returned after a successful authentication should also be verified:

- When the authenticated token is retrieved from the Security Context, the user properties (username, credentials) should be consistent with initialization values when retrieved

Scenario: Token retrieval attempted from storage mechanism

Applications using the OAuth 2.0 mechanism should test *token retrieval* behaviors to ensure tokens are only returned with correct inputs and requests for missing tokens are handled:

- When *valid* user credentials are provided to retrieve a user's access tokens, the correct tokens are returned
- When *invalid* user credentials are provided to retrieve a user's access tokens, tokens are not returned
- When an access token is requested which *does not exist*, the application handles this failure (e.g. throws Exception) and does not return a token

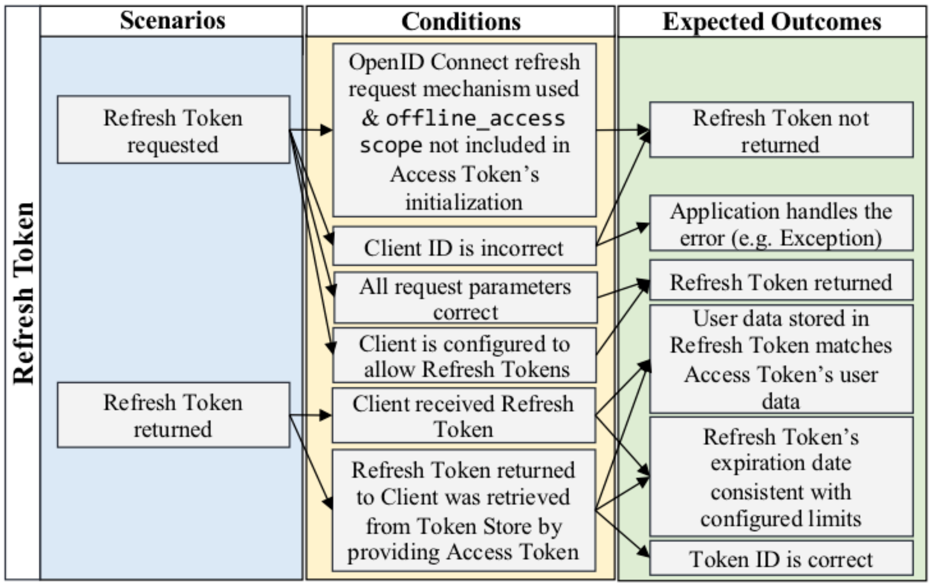
Scenario: Token passed to validator

Most security APIs provide token validation techniques, or a custom solution can be built. All validation mechanisms should be tested to ensure that attempts fail when:

- The token value provided is null
- The token value provided is empty (string)
- The token type is unsupported
- The token is expired
- Only part of the token is provided

Tests for Refresh Tokens (OAuth 2.0)

Applications that allow authentication via OAuth 2.0 should verify that refresh token properties are correct and creation/access behaviors succeed only for valid inputs.


Refresh Token
Scenarios
Conditions
Expected Outcomes
Refresh Token requested
Refresh Token returned
OpenID Connect refresh request mechanism used & offline_access scope not included in Access Token's initialization
Client ID is incorrect
All request parameters correct
Client is configured to allow Refresh Tokens
Client received Refresh Token
Refresh Token returned to Client was retrieved from Token Store by providing Access Token
Refresh Token not returned
Application handles the error (e.g. Exception)
Refresh Token returned
User data stored in Refresh Token matches Access Token's user data
Refresh Token's expiration date consistent with configured limits
Token ID is correct

Scenario: Refresh token requested

Requests for refresh tokens should be tested to ensure that they are only when allowed:

- When the OpenID Connect refresh request mechanism is implemented, and the `offline_access scope` is not included in the access token's initialization, a Refresh Token should not be returned when requested
- If the client is configured to allow refresh tokens, then the access token should return a valid refresh token when requested
- If a refresh token is requested but the Client ID is incorrect, an Exception (e.g. `InvalidTokenException`) should be thrown

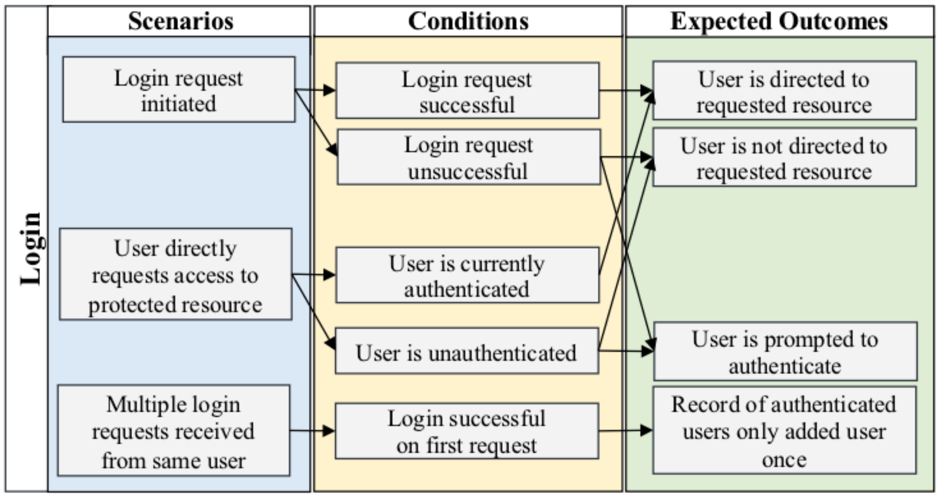
Scenario: Refresh token returned

Properties of the refresh token should be verified for consistency with the original access token:

- The refreshed token should contain the same principal
- The refreshed token should contain the same credentials
- The refreshed token's expiration date should be consistent with configured validity limits
- When an access token is provided to retrieve a refresh token from a token store, the returned token ID is correct

Tests for Login Form Submission

Alongside testing the functionality of authentication mechanisms, it is important to test an application's user-facing behavior after a login form is submitted by the user.


Login
Scenarios
Conditions
Expected Outcomes
Login request initiated
User directly requests access to protected resource
Multiple login requests received from same user
Login request successful
Login request unsuccessful
User is currently authenticated
User is unauthenticated
Login successful on first request
User is directed to requested resource
User is not directed to requested resource
User is prompted to authenticate
Record of authenticated users only added user once

***Scenario*: Login request initiated**

When a user submits a login request, they typically expect to be redirected to a resource "inside" the application (e.g. your personal Facebook timeline).

Tests should ensure this only occurs when authentication succeeds:

- If the login request succeeds, the user should be redirected to the default "interior" page
- If the login request fails:
  - The user should *not* be redirected to the requested resource
  - The user should be prompted to authenticate in some way, such as redirecting to login page or completing a pop-up login form.

***Scenario*: User directly requests access to a protected resource**

Sometimes a user tries to navigate directly to a protected page from the browser. For example, to access a Facebook profile you might open your browser and type `facebook.com/myProfile` instead of navigating there from the homepage (`facebook.com`). In this situation, the application should be tested to ensure that it verifies the user's authentication status before returning the requested resource.

If the user is not authenticated:

- The application should not redirect to the requested resource
- The user should be prompted to authenticate in some way, such as redirecting to login page or completing a pop-up login form.

If the user is authenticated, the application should redirect to the requested resource.

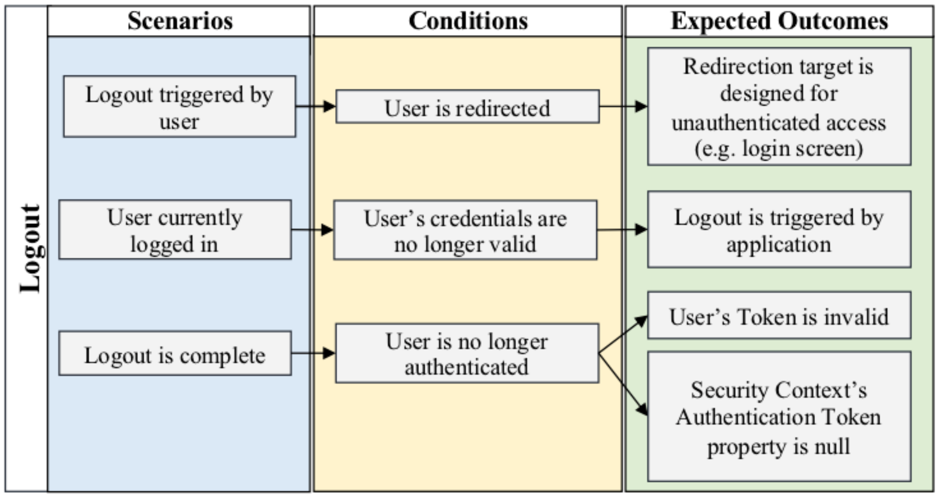
Scenario: Multiple login requests received from the same user

If a user clicks the submit button repeatedly, multiple login requests may be sent to the server. Tests should verify the application's state is not redundantly updated in this scenario:

- If authentication succeeds after multiple duplicate requests, the record of currently authenticated users should only contain one record for the user.

Tests for Logout

A logout even can be triggered directly by a user, or the application can self-initiate based on a change in the user's authenticated status. Tests should verify that these events are correctly initiated, and the application's state is appropriated after logout is complete.


Logout
Scenarios
Conditions
Expected Outcomes
Logout triggered by user
User is redirected
Redirection target is designed for unauthenticated access (e.g. login screen)
User currently logged in
User's credentials are no longer valid
Logout is triggered by application
Logout is complete
User is no longer authenticated
User's Token is invalid
Security Context's Authentication Token property is null

Scenario: Logout triggered by user

The user should be immediately redirected to a part of the application designed for unauthenticated access (e.g. login or "home" page)

Scenario: User currently logged in

The application should regularly verify the authentication status of active users and initiate a logout if a user's authentication details (e.g. credentials) are no longer valid

Scenario: Logout is complete

- The Security Context's authenticated token property should be null
- The user's token should no longer be valid
- The user's token should have been removed from the Token Store

A Summary of Relevant Unit Test Smells to Avoid

The test cases in this guide were derived from real unit tests written by open source developers. In fact, to construct this guide, over 400 unit test methods from more than 50 open source projects were manually analyzed.

During this analysis, several unit test smells were repeatedly observed across scenarios and features. This section provides brief descriptions of these smells and any "links" between a smell and a specific scenario/feature.

It is recommended to avoid these smells in *any* unit tests, but these specific smells are included because they seem to appear often in token authentication unit tests. An excellent reference with detailed descriptions of each smell, it's causes, and mitigations is *XUnit Test Patterns: Refactoring Test Code* by Gerard Meszaros.

Obscure Test This smell describes difficulties with test comprehension. It commonly stems from an "eager" implementation, where the test verifies too much functionality. For example, instead of providing dedicated test cases, the developers decided to combine similar behaviors ("piggyback'') into one long ("the giant'') test case. This results in higher maintenance costs, difficulties in understanding the test, and precludes achieving the goal to use *tests as documentation*. It also may hide many more errors, because during test execution, the first failed assertion aborts the test case.

Many obscure tests were observed for the Authenticating with Tokens feature. Particularly, the same test case would attempt to authenticate as anonymous user, as guest user, as administrator and as user with a disabled account. This all happens in sequence instead of writing dedicated test cases for each user type.

Conditional Test Logic This smell occurs when the test case contains code that may or may not be executed. It complicates reasoning about the actual authentication code and the path taken during test execution; which branch was taken, or which iteration failed.

Several instances of this smell were observed in tests for the *Token passed to authenticating servlet filter via HTTP request* scenario. For example, some tests traversed different configurations in a loop, so the action performed during the test was dependent on the current configuration.

Test Code Duplication In this smell, the same test code is repeated many times. Duplicated code impacts test code maintenance, which can be a serious issue: if test logic in on code fragments is modified, all duplicates must be identified & modified as well.

Two manifestations of this were observed: either whole test cases were duplicated (with slight modifications), or code chunks within a test case were repeated. This smell mostly in occurred in test cases for authentication protocols, e.g. SAML or OAuth. Instead of parameterizing the test cases, the whole test case is duplicated with minor changes, e.g. testing OAuth with different identity providers like Google or Facebook.

Assertion Roulette This smell (also called the "free ride"), is similar to *Obscure Test*; the key difference is the test case is not necessarily long or complex, but always includes multiple assertions. While this is not always an issue, if these assertions do not have *messages,* it is hard to tell which assertion caused the test to fail. This smell was commonly found in tests for the Token Authentication and Manipulation features. A single test may have assertions for multiple user types credentials such as unknown user, normal user, and administrator. Another example is one test for multiple user detail conditions such as different kinds of invalid passwords.

Excessive Mocking This “mock happy'' smell refers to a test case which requires many mocked components to run. The system under test may need a complex environment, specific resources, or depend on 3rd party systems which cannot be provided during test execution.

Mocks are a valuable asset for unit testing authentication, but sometimes it can be difficult to determine the appropriate mocking strategy. For example, one developer posted the question “Spring Test & Security: How to mock authentication?” to Stack Overflow: they wanted to write unit tests to verify that URLs of the developed web service are properly secured. Spring Security has a built-in solution for using mocks in tests, which was also picked as answer for the referenced question. However, use of these built-in solutions in practice were rarely observed. Instead, developers came up with individual ones leveraging 3rd party mocking frameworks. Further, some tests had an overwhelming number of mocked objects and behaviors, which made it difficult to understand what was actually “real'' and being tested.

Issues in Exception Handling “Silent catcher'' test cases pass even when an exception is thrown which may lead to false positive results. Typically, the exception is either ignored entirely or the *type* of exception is not considered. For example, consider a test whose intended behavior is to check whether an exception is thrown when invalid credentials are provided. Without detailed handling of the exception, the test may pass when an unexpected type of exception is thrown, such as if the database storing the users is not available.

Testing the Authentication Framework This smell is previously un-named but was observed in the open source test dataset. Sometimes developers write tests for functionality that is fully implemented by the API (instead of testing custom implementations of API-provided interfaces). For example, a test case was observed to validate the correct behavior of the SHA-1 cryptographic hash implementation provided by the Spring framework. The test literally compares the generated message digest with a string constant to see if both are identical. Since the hash implementation is part of a 3rd party library, the library maintainers are responsible to properly test the code and not the developer.

Mocking a Mocking Framework Multiple test cases that tested the applied mocking framework were also observed. There, the test author assembles a specific object, e.g. a user, and instructs the mock library to return this object when the function under test is called. After performing this call, it is asserted whether the returned object is identical to the one the mock library was instructed to return. Thus, the correct behavior of the mock framework is tested and not that of the function.